RADC-TR-65-377, Vol. II
Final Report

INFORMATION SYSTEM THEORY PROJECT

Volume II

Collected Research Papers

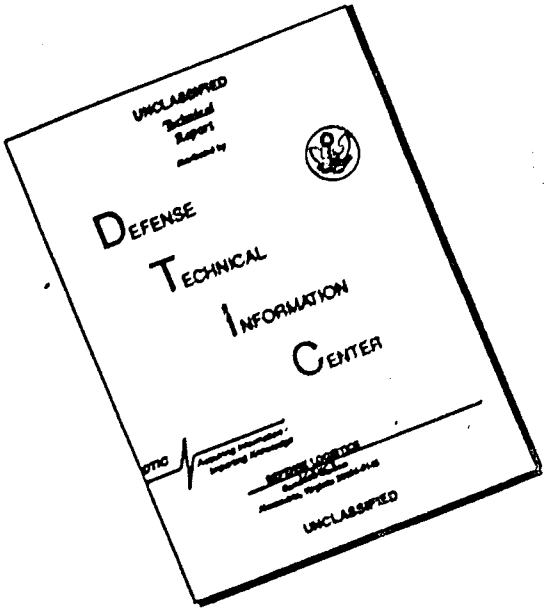
Thomas E. Cheatham, Carlos Christensen
Robert W. Floyd, et al
Computer Associates, Inc.

TECHNICAL REPORT NO. RADC-TR-65-377

November 1965

Reconnaissance–Intelligence Data Handling Branch
Rome Air Development Center
Research and Technology Division
Air Force Systems Command
Griffiss Air Force Base, New York

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

INFORMATION SYSTEM THEORY PROJECT

Volume II

Collected Research Papers

Thomas E. Cheatham, Carlos Christensen
Robert W. Floyd, et al
Computer Associates, Inc.

FOREWORD

The papers collected in this report represent research results developed within the Information System Theory Project by various staff members of Computer Associates, Inc. The work reported was supported in part by the Rome Air Development Center under Contract AF 30 (602)-3324.

This report has been reviewed and is approved.

Approved: Patricia M. Langendorf
PATRICIA M. LANGENDORF
Project Engineer

Approved: Howard Davis
ROBERT J. QUINN, Jr. Colonel, USAF
Chief, Intel and Info Processing Div

FOR THE COMMANDER: Irving J. Gabelman
IRVING J. GABELMAN
Chief, Advanced Studies Group

TABLE OF CONTENTS

NOTES ON COMPILING TECHNIQUES

by

T. E. Cheatham, Jr.

CA-6505-0611

This document consists of notes on a series of lectures to be given at the University of Michigan Summer Conference Course on Automatic Programming, June 1965.

Preface

The construction of compilers for the variety of programming languages and computing machines required today has become a major activity in the programming field. The amount of published and readable material on the subject of compiler construction is, however, very limited. There are many reasons for this:

1. Relatively few academic types have been interested in the problem until recently.

2. There is a rather basic lack of any theoretical foundations for much of the manipulation required in a compiler -- the specification and manipulation of a formal syntax of a programming language being, of course, an exception.

3. There have been relatively few "projects" undertaken with the idea of developing general purpose compiling techniques; the general techniques which have emerged in spite of this orientation have often not been apparent.

4. Many of the techniques developed are documented in a form that is so specialized to a certain programming language or computer that the publication in anything but a compiler maintenance manual would be deemed unthinkable.

The intention of these notes is to provide a sketch of several basic compiling techniques. It is hoped that the approach and method will point toward a general framework for a compiler. It is certainly not the authors claim that this comprises a theory of compiling or even a good framework. All we can hope is that the presentation is such that the reader can gain a basic understanding of a variety of isolated techniques and that a glimmer of how to tie them together might result.

Another caveat: we have been at the business of programming sufficiently long to have changed our ideas about computers, programming, compiling, languages, etc. sufficiently often to be aware that our current ideas about the techniques and method of presentation provided here will undoubtedly change. Thus, accept these notes as our ideas circa summer 1965.

1. General Discussion

In very broad terms, we will talk about the compilation process as consisting of four phases (stages, activities, etc.), namely:

parsing* the input string;
interpreting the parse of the input string;
analysis of the computation sequence and generation of machine coding; and
contacting the environment in which the compiler and/or resulting code is to operate

We must emphasize that we do not consider these four phases as "the four phases" of a compiler. Rather, these simply represent a convenient partitioning of the compilation activity for discussion purposes. Sections 2. through 5. are devoted to a discussion of these four phases.

Section 6. is devoted to a discussion of several miscellaneous topics concerning languages and compilers generally.

Section 7. contains some remarks about programming systems -- as environments for a compiler and its' product.

2. Parsing the Input

2.0. General Discussion

In very general terms, parsing the input source statements in some programming language amounts to the following: We are given a stream of characters comprising a source program. We first perform a lexical analysis, i.e., we isolate, identify and tabulate the sub-strings corresponding to certain "elementary components" in the source program. These elementary components may be the basic symbols of the source language (in which case the transformation of the input stream is rather slight -- amounting only to recognizing and isolating sub-strings corresponding to operators or delimiters [BEGIN, EQ, (,etc.]) or they may

---

* i.e., analyzing and restructuring the input string in such a fashion that interpretation and subsequent processing is facilitated.

be the literals, identifiers, operators, and delimiters of the source language (in which case the transformation is somewhat more interesting). In any event, we will think of the lexical analysis as a process of transforming the initial input stream of characters into a stream of "descriptors". A descriptor, in the sense in which we shall employ it, is a pair (TABLE, LINE) denoting the table containing the thing (symbol) described and the line within the table corresponding to the thing described. Thus, TABLE might indicate symbol table, literal table, terminal symbol table, character table, and so on.

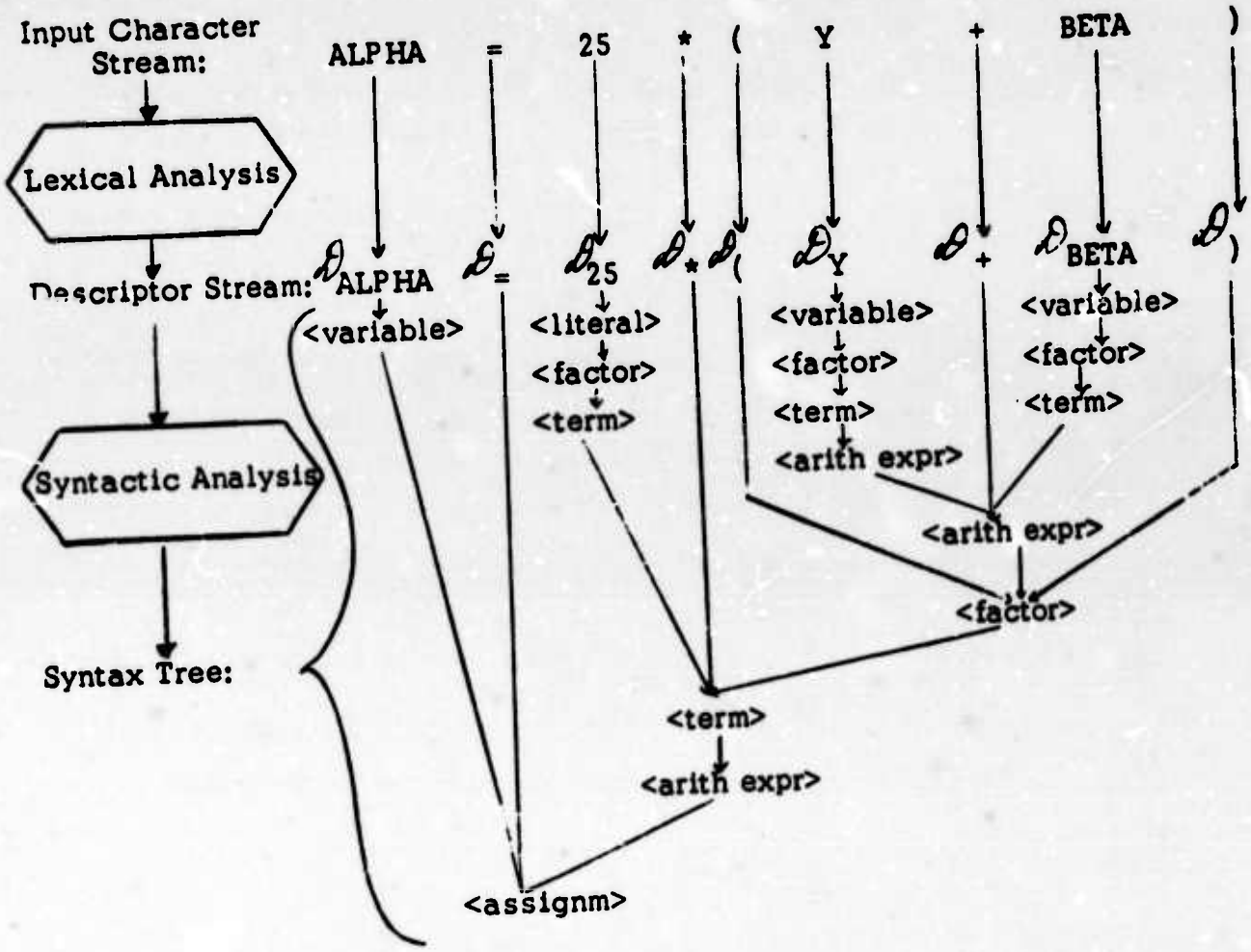
Given this stream of descriptors we then perform a syntactic analysis -- i.e., we isolate, identify, and tabulate the substrings corresponding to the syntactic structures in the source program. We will view this process in two different ways, as follows:

1. Producing a "tree" which represents the complete syntactic analysis of (some portion of) the source program.
2. Producing a list of "phrases" which represent a partial syntactic analysis of (some portion of) the source program.

This is not to say that a syntactic analysis results either in "trees" or in a list of "phrases"; indeed some of the most interesting parsing procedures result in a mixture of these two extremes of output.

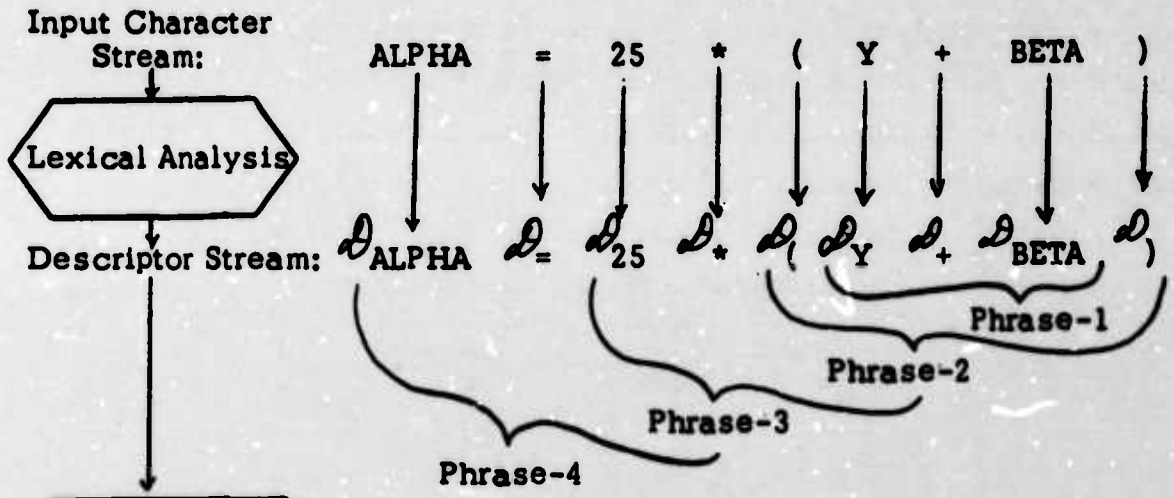
The following examples might make these less vague:

Analysis of Source into a Syntax Tree



Note: For $\mathscr{D}_{ALPHA}$, $\mathscr{D}_{=}$, $\mathscr{D}_{25}$ and so on read descriptor of the symbol ALPHA, descriptor of the terminal character =, descriptor of the (integer) literal 25.

Analysis of Source into a List of Phrases



In the section following, we will discuss several aspects of the parsing of source statements. Section 2.1 deals with the specification of syntax of a programming language; we will see later in the sections on "syntax directed" analysis that it is possible to directly utilize a syntax specification to parse source statements. Section 2.2 is concerned with lexical analysis and section 2.3 deals with syntactic analysis. Section 2.4 deals with the problems of error analysis and error recovery with various kinds of syntactic analysis and, finally, section 2.5 discusses the problem of mechanically generating a syntax analyzer from a formal syntax specification.

2.1. Syntax Specifications

Several essentially equivalent formalisms for the representation of syntax have been developed. These include such things as

Post Production Systems, developed by the logician Emit Post during the 1940's as a tool in the study of Symbolic Logic.

Phrase Structure Grammars, developed by the linguist Noam Chomsky during the 1950's as a tool in the study of natural languages.

Backus Normal Form, developed by the programmer John Backus during the late 1950's as a tool in the description of programming languages.

Throughout these notes we will utilize a formalism similar to Backus's. An appendix* contains a reasonably detailed account of our particular variation. For the present, however, we will present only a few highlights by giving a syntax for a very simple programming language which we will utilize throughout much of the following discussion.

2-13

A syntactic specification of a language is a concise and compact representation of the structure of that language, but it is merely that - a description of structure - and does not by itself constitute a set of rules either for producing allowable strings in the language, or for recognizing whether or not a proffered string is, in fact, an allowable string. However, rules can be formulated to produce, or recognize, strings according to the specification.

In order to discuss the structure of the language, we give names to classes of strings in the language - we call these names (or the classes they denote) syntactic types. Some of the classes of interest consist of fixed strings of characters of the source alphabet: these we call terminal types, and specifically terminal symbols; to talk about any particular one, we will merely display the character string. Most of the classes, though, are more complicated in structure and are defined in terms of other classes; these we call defined types, and to designate one, we choose a mnemonic name for the class and enclose it in the signs '<' and '>'.

---

* See Cheatham & Sattley, "Syntax Directed Compiling".

A simple type definition consists of the name of a defined type, followed by the curious sign '::=' followed by a sequence of syntactic types, defined or terminal. An example, taken from the syntax I -- soon to be discussed -- would be*:

<ustat>::=<ident>=<ae>

In general we shall call the right-hand side of the definition the definiens. Any sequence of type designators appearing in a definiens is called a construction, and each type designator within the construction is a component of the construction. So, the above example is a definition of the defined type <ustat>; its definiens is a construction with three components, which are, in the order of their appearance, the defined type <ident>, the terminal character '=' and the defined type <ae>.

A simple type definition of this sort states that, among the strings of the source language belonging to the defined type, are those which are concatenations of substrings -- as many substrings as there are components of its (simple) definiens -- such that each substring (in order of concatenation) belongs to the syntactic type named by the corresponding component (in order of appearance in the definiens). Applied to our example: A source string belongs to the class <ustat> (or, for short, "is an <ustat>") if it can be broken into three consecutive substrings, the first of which is a <ident>, the second of which is the single character '=', and the third of which is an <ae>.

As another example, the rule

<integer>::= <digit>| <integer><digit>

can be read: A source string belongs to the class <integer> if it is either a

---

* Read: ustat ≡ unconditional statement; ident ≡ identifier; ae ≡ arithmetic expression.

<digit> or if it can be broken into two substrings, the first of which is an <integer> and the second of which is a <digit>. Thus, the "|" is read as "or". The constructs <digit> and <integer> <digit> are called the alternatives of the defined type <integer>.

Following are two syntax specifications for a simple programming language, language L_T which we will reference throughout these notes:

Syntax I for Language L_T

1. <letter> ::= A| B| C| D| E| F| G| H| I| J| K| L| M| N| O| P| Q| R| S| T| U| V| W| X| Y| Z
2. <digit> ::= 0| 1| 2| 3| 4| 5| 6| 7| 8| 9
3. <ident> ::= <letter>| <ident> <letter>| <ident> <digit>
4. <integer> ::= <digit>| <integer> <digit>
5. <rlop> ::= LT| LE| EQ| UE| GE| GT
6. <adop> ::= +|-
7. <factor> ::= <ident>| <integer>|(<ae>)
8. <term> ::= <factor>| <term>*<factor>
9. <ae> ::= <term>| <ae> <adop> <term>
10. <rel> ::= <ae> <rlop> <ae>
11. <cstat> ::= <rel> THEN <ustat>| <rel> THEN <ustat> ELSE <ustat>
12. <ustat> ::= <ident> = <ae>| TO <ident>|.STOP
13. <stat> ::= <ustat> .| <cstat> .| <ident>..<stat>
14. <prog> ::= <stat>| <prog> <stat>

2-14

2-15

Syntax II for Language L_T

3. <ident> ::= *v*
4. <integer> ::= *d*
5. <rlop> ::= LT | LE | EQ | UE | GE | GT
6. <adop> ::= + | -
7. <factor> ::= <ident> | <integer> | (<ae>)
8. <term> ::= <factor> | <term>*<factor>
9. <ae> ::= <term> | <ae> <adop> <term>
10. <rel> ::= <ae> <rlop> <ae>
11. <cstat> ::= <rel> THEN <ustat> | <rel> THEN <ustat> ELSE <ustat>
12. <ustat> ::= <ident> = <ae> | TO <ident> | STOP
13. <stat> ::= <ustat> . | <cstat>. | <ident>..<stat>
14. <prog> ::= <stat> | <prog> <stat>

These two syntaxes differ in that the first assumes we have a recognizer for the basic characters only; the second assumes we have a recognizer for substrings which we choose to consider as members of the classes <ident> and <integer> respectively. That is, we are assuming that the lexical analyzer is competent to isolate and deal with identifiers and literals.

Consider the string "Z = 1."; we can "prove" that this string is a <prog> according to syntax I for language L_T as follows:

The	construct	Z	is	of	class	<letter>	by	rule	1,	alternative	26.
"	"	Z	"	"	"	<ident>	"	"	3,	"	1.
"	"	1	"	"	"	<digit>	"	"	2,	"	2.
"	"	1	"	"	"	<integer>	"	"	4,	"	1.
"	"	1	"	"	"	<factor>	"	"	7,	"	2.
"	"	1	"	"	"	<term>	"	"	8,	"	1.
"	"	1	"	"	"	<ae>	"	"	9,	"	1.
"	"	Z= 1	"	"	"	<ustat>	"	"	12,	"	1.
"	"	Z= 1.	"	"	"	<stat>	"	"	13,	"	1.
"	"	Z= 1.	"	"	"	<prog>	"	"	14,	"	1.

Another <prog> in language L_T is:

2-16

2-17

```
X = 1.      Y = 0.
HENRY..IF X LT 10 THEN Y = Y*Y ELSE TO HENRY. STOP.
```

2.2. Lexical Analysis

It is, in principle, possible to "bypass" a specific lexical analysis phase in parsing source programs. Indeed several compilers* perform this function as part of the overall syntactic analysis. However, when it comes to writing a compiler in practice, the question of recognizing terminal characters in a language (e.g., "+", "EQ", etc.) brings us face to face with the many problems of restricted character sets, input/output idiosyncracies of computers, and so on. Further, unless the resulting code is to be handed on in symbolic form to some assembly processor, the identifiers and literals in the source language have to be replaced by (probably relocateable) addresses in the resulting code and the contents of addresses allocated to the literals must contain the appropriate binary (decimal, etc., depending on the machine) representation. Thus, it becomes convenient to postulate a lexical analyzer interposed between the source of input characters and the syntactic analyzer. Further, with languages such as FORTRAN II and FORTRAN IV which utilize fixed fields for indicating labels, end of statement, and the like, it makes the syntactic analysis "cleaner" if label and end of statement delimiters are inserted into the source before syntactic analysis commences.

For most of these notes we will postulate that a lexical analysis is performed on an input stream of characters resulting in the isolation of identifiers, literals, and the operators and delimiters in the source language. We further postulate that the identifiers are placed in a symbol table and that the lexical analyzer outputs an appropriate descriptor. Further, that literals are placed in a literal table with the output of an appropriate descriptor. Finally, we presume the existence of a terminal symbol table containing (initially) entries for all operators and delimiters in the source language and assume that the lexical analyzer isolates all such operators and delimiters, returning a descriptor pointing to the appropriate terminal symbol table entry.

---

* See, for example, E. T. Irons, "A Syntax Directed Compiler for ALGOL-60", Communications of the ACM, Vol. 4, pp 51-55, January 1961.

2.3. Syntactic Analysis

We will discuss in some detail two basically different types of syntactic analysis,

> Predictive analysis, with "top down" and "bottom up" variations; and
>
> Bounded context analysis, using the "precedence" technique and the "productions" technique.

We will further discuss one "mixed" (i.e., predictive and bounded context) method of analysis and remark briefly on several variations of all these techniques.

2.3.1. Predictive Analysis

2-19

2-18

Very broadly, predictive analysis works as follows: Given a set of rules (a syntax specification) for forming allowable constructs in a language, eventually resulting in a program* in that language, we analyze a source statement by guessing or predicting how the statement is constructed and either verifying that this is the case or backing up to try again, assuming some other method of construction. We keep a "history" of our attempts and when we have determined the exact way in which the program is constructed we can use this "history" to produce the complete syntactic analysis (tree).

The "top down" variation of predictive analysis is roughly the following: The analyzer has, at any point in time, a current goal. At the beginning of the process, it takes the starting type** of the specification as its' first goal. Then at any point in the process it follows these steps when it has a defined type as its current goal:

---

* or whatever is the "largest" syntactic type in the language

** i.e., the "largest" syntactic types; in the case of language L_T this is <prog>; see the Cheatham, Sattley paper in the appendix for details.

The analyzer consults the definition of the defined type (in Backus Normal Form each defined type has a unique definition), and, specifically, it considers the first alternative in that definition. It then successively takes each component of that alternative as a sub-goal. (Of course, it must re-enter itself for each of these goals, and it must keep track of where it was at each level of re-entry.) If at any point it fails to find one of these sub-goals, it abandons that alternative, and considers the next alternative in that definition, if there is one, and steps through the components of that alternative. If there is no next alternative, it has failed to realize its current goal, and reports this fact "upstairs". If it succeeds in finding the sub-goals corresponding to each of the components of any alternative in the definition of its current goal, it has found its goal, and reports that fact.

This rough sketch conventiently ignores a number of sticky points (which we will consider via an example in section 2.3.1.1) but should serve to give a rough idea of the top down analysis process.

The "bottom up" variation of predictive analysis is a bit more difficult to describe. As with the "top down" process, the "bottom up" process has at any time a current goal (initially the starting type of the syntax specification). The analyzer reads an input descriptor (or character, if there is no lexical analyzer interposed between the input character stream and the syntactic analyzer) and checks to see whether it has "gone astray" in trying to reach its goal or whether the syntactic type of the input is a possible first component of a first component of a ... of the goal. If so it proceeds to read more input and progress toward the goal; if not, it continues processing input until it has built another syntactic type of which the previous one is a first component and goes back to the checking. If it has gone astray, it backs "down" and tries to see if it can construe the input differently, to approach its' goal along a different chain of intermediate types.

Again this description is vague and we will return to a detailed consideration of bottom up analysis later, via an example in section 2.3.1.2 and in detail in an appendix.

To recap "top down" versus "bottom up": "top down" sets a goal and tries all possible ways of achieving that goal before giving up and replacing the goal with an alternative; "bottom up" has a goal and "reads" input trying to construe the input as a first component of .... of the goal and to continue reading input to realize the goal through the chain of intermediate syntactic types implied. Thus, in a top down analysis, the syntax drives the search; in bottom up analysis, the input drives the search.

2.3.1.1. Example of Top Down Predictive Analysis

The following table depicts the course of a top down predictive analysis with the input descriptor stream.

$$\mathscr{D}_{\text{ALPHA}}\ \mathscr{D}_{=}\ \mathscr{D}_{25}\ \mathscr{D}_{*}\ \mathscr{D}_{(}\ \mathscr{D}_{\text{Y}}\ \mathscr{D}_{+}\ \mathscr{D}_{\text{BETA}}\ \mathscr{D}_{)}\ \mathscr{D}_{.}$$

The column headings and their interpretations are as follows:

2-20

2-21

Step- obvious

Goal- current goal

Next descriptor- The next descriptor available on the input stream; we will drop the " " for readability.

Current rule- The number of the syntax rule being applied (line number in Syntax I or Syntax II as given in Section 2.1).

Current alternate- Alternate construction currently being applied (numbered 1, 2 ...)

Current component- Component (of alternate of rule) currently being applied.

The process starts with "prog" as initial goal.

Step	Goal	Next Descriptor	Current Rule	Alt.	Comp.	Remarks
1	prog	ALPHA	14	1	1	Set sub-goal, stat
2	stat	ALPHA	13	1	1	Set sub-goal, ustat
3	ustat	ALPHA	12	1	1	Set sub-goal, ident
4	ident	ALPHA	3	1	1	Recognize; consider next input and return to previous goal (3), next component
5	ustat	=	12	1	2	Set sub-goal, =
6	=	=	-	-	-	Recognize; consider next input and return to previous goal (5), next component
7	ustat	25	12	1	3	Set sub-goal, ae
8	ae	25	9	1	1	Set sub-goal, term
9	term	25	8	1	1	Set sub-goal, factor
10	factor	25	7	1	1	Set sub-goal, ident
11	ident	25	3	1	1	Fail; return to previous goal (10), next alternative
12	factor	25	7	2	1	Set sub-goal, integer
13	integer	25	4			Recognize; consider next input and return to previous goal (12), next component
14	factor	*	7	2	-	Succeed; return to previous goal (9), next component
15	term	*	8	1	1	Succeed; however, since <term> is defined in terms of <term>, try for a "larger" term;
16	term	*	8	2	2	Set sub-goal, *
17	*	*				Recognize; consider next input and return to previous goal (16), next component
18	term	(	8	2	3	Set sub-goal, factor

Step	Goal	Next Descriptor	Current Rule	Alt.	Comp.	Remarks
19	factor	(	7	1	1	Set sub-goal, ident
20	ident	(				Fail; return to previous goal, (19), next alternative
21	factor	(	7	2	1	Set sub-goal, integer
22	integer	(				Fail; return to previous goal, (21), next alternative
23	factor	(	7	3	1	Set sub-goal, (
24	(	(				Recognize; consider next input and return to previous goal (23), next component
25	factor	Y	7	3	2	Set sub-goal, ae
26	ae	Y	9	1	1	Set sub-goal, term
27	term	Y	8	1	1	Set sub-goal, factor
28	factor	Y	7	1	1	Set sub-goal, ident
29	ident	Y				Recognize; consider next input and return to previous goal (28), next component
30	factor	+	7	1	1	Succeed; return to previous goal (27), next component
31	term	+	8	1	1	Succeed; however since <term> is defined in terms of <term>, try for larger <term>
32	term	+	8	2	2	Set sub-goal, *
33	*	+				Fail; return to previous goal (32), next alternative
34	term	+				Failure meant only can't build larger term; return to previous goal (26), next component
35	ae	+	9	1	1	Succeed; however, since <ae> is defined in terms of <ae>, try for a "larger" <ae>.

2-22

2-23

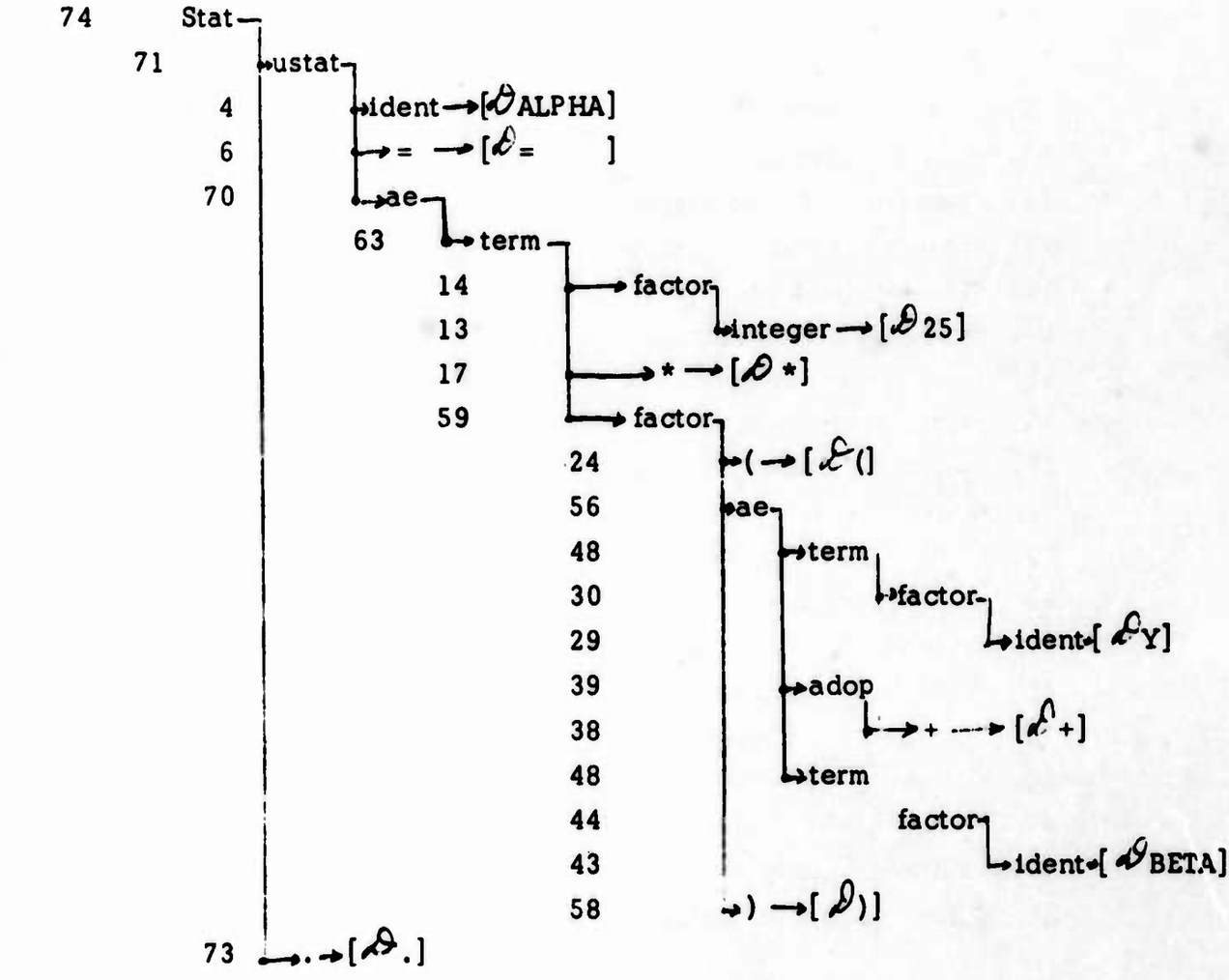
Step	Goal	Next Descriptor	Current Rule	Alt.	Comp.	Remarks
36	ae	+	9	2	2	Set sub-goal, adop
37	adop	+	6	1	1	Set sub-goal, +
38	+	+				Recognize; consider next input and return to previous goal (37), next component
39	adop	BETA	6	1	1	Succeed; return to previous goal (36), next component
40	ae	BETA	9	2	3	Set sub-goal, term
41	term	BETA	8	1	1	Set sub-goal, factor
42	factor	BETA	7	1	1	Set sub-goal, ident
43	ident	BETA				Recognize; consider next input and previous goal (42), next component
44	factor	)	7	1	1	Succeed; return to previous goal (41), next component
45	term	)	8	1	2	Succeed; however since <term> is defined in terms of <term>, try for "larger" <term>
46	term	)	8	2	2	Set sub-goal, *
47	*	)				Fail; return to previous goal (46)
48	term	)	8	2	2	Failure means only can't build a larger <term>; return to previous goal (40), next component
49	ae	)	9	2		Succeed; however since <ae> is defined in terms of <ae> try for "larger" <ae>.
50	ae	)	9	2	2	Set sub-goal, adop
51	adop	)	6	1	1	Set sub-goal, +

2-24

Step	Goal	Next Descriptor	Current Rule	Alt.	Comp.	Remarks
52	+	)				Fail; return to previous goal (51), next alternative
53	adop	)	6	2	1	Set sub-goal, -
54	-	)				Fail, return to previous goal (53), next alternative
55	adop	)	6	-	-	Fail; return to previous goal (54), next alternative
56	ae	)				Failure means only can't build larger <ae>; return to previous goal (25), next component
57	factor	)	7	3	3	Set sub-goal,)
58	)	)				Recognize; return to previous goal (57), next component
59	factor	.	7	3	3	Success; return to previous goal (18), next component
60	term	.	8	2	-	Success; however since <term> is defined in terms of <term>, try for a larger <term>
61	term	.	8	2	2	Set sub-goal, *
62	*	.				Fail; return to previous goal next alternative
63	term	.				Failure means only can't build larger <term>; return to previous goal (8), next component
64-69	ae	.	9	1	-	Success; here we repeat essentially steps 50-55
70	ae	.				Failure means only can't build larger <ae>; return to previous goal (7), next component

Step	Goal	Next Descriptor	Current Rule	Alt.	Comp.	Remarks
71	ustat	.				Succeed; return to previous goal (2), next component
72	stat	.	13	1	2	Set sub-goal, .
73	.	.				Recognize; return to previous goal (1), next component
74	prog	empty				Out of input; have succeeded.

We note that the successful recognitions allow us to trace the syntax tree as follows:



2.3.1.2. Examples of Bottom Up Predictive Analysis

In order to utilize syntax II for bottom up analysis it is convenient to rewrite the rules "in reverse" as follows:

1. $\mathcal{V}$ =:: <ident>
2. $\mathcal{I}$ =:: <integer>
3. LT =:: <rlop>
4. LE =:: <rlop>
5. EQ =:: <rlop>
6. UE =:: <rlop>
7. GE =:: <rlop>
8. GT =:: <rlop>
9. + =:: <adop>
10. - =:: <adop>
11. <ident> = <ae> =:: <ustat>
12. <ident> .. <stat> =:: <stat>
13. <ident> =:: <factor>
14. <integer> =:: <factor>
15. (<ae>) =:: <factor>
16. <factor> =:: <term>
17. <term> * <factor> =:: <term>
18. <term> =::<ae>
19. <ae> <adop> <term> =:: <ae>
20. <ae> <rlop> <ae> =:: <rel>
21. <rel> THEN <ustat> ELSE <ustat> =:: <cstat>
22. <rel> THEN <ustat> =:: <cstat>
23. TO <ident> =:: <ustat>
24. <ustat> . =:: <stat>
25. <cstat> . =:: <stat>
26. <stat> =:: <prog>
27. <prog> <stat> =:: <prog>

In bottom up analysis we are interested, given a syntactic type or an input symbol, "where can it lead?". Thus, having recognized an <ident> we can see by rules 11 - 13 that we can be led to a <ustat> a <stat> or a <factor> directly* by

---

* and, to a <stat> (by 24) a <prog> (by 26), a <term> (by 16) (by one level) indirectly, and so on.

finding an = followed by an <ae>; a .. followed by a <stat>; or nothing following respectively, etc.

In order to show the bottom up analysis, it is convenient to shrink our notation somewhat. First, we denote the various syntactic types by lower case letters as follows:

Type	Symbol
<stat>	s
<ustat>	u
<ae>	a
<term>	t
<factor>	f
<ident>	i
<integer>	n

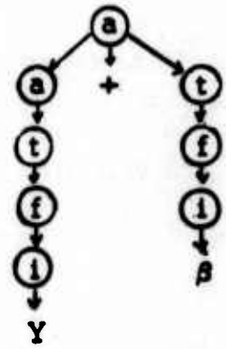
We rewrite the string* to be analyzed in the more compact form

$$\alpha = 25 * (Y + \beta)$$

We further introduce a bracket notation to indicate syntactic structure; thus:

i[α], i[Y], i[β] indicate that α, Y, β are of syntactic type i (=<ident>)

a[a[t[f[i[Y]]]]+ t[f[i[β]]]] indicates the tree structure



and so on.

We rewrite the above syntax in the following form.

No.	Rule
1-10	Not used in the example
11	i[ ] → u[ ]▲=a]
12	i[ ] → s[i[ ]▲..s]
13	i[ ] → f[i[ ]]
14	n[ ] → f[n[ ]]
15	(→ f[(▲a)]
16	f[ ] → t[f[ ]]
17	t[ ] → t[t[ ]▲*f]
18	t[ ] → a[t[ ]]
19	a[ ] → a[a[ ]▲+t]
20	a[ ] → r[a[ ]▲rt a]
21-27	Not used

Read (for rule 11): "If an i[ ] appears on the input stream then it can be the first component of a u; the pointer ▲indicates that an "=" followed by an "a" must be found following the i[ ] in order to complete the recognition of such a u."

We presume that the lexical analysis of the source string has produced the following (descriptors):

$$\mathcal{D}_{i[\alpha]}\,\mathcal{D}_{=}\,\mathcal{D}_{n[25]}\,\mathcal{D}_{*}\,\mathcal{D}_{(}\,\mathcal{D}_{i[\gamma]}\,\mathcal{D}_{+}\,\mathcal{D}_{i[\beta]}\,\mathcal{D}_{)}$$

That is, we assume that the syntactic type of each identifier and integer has been "attached" to their descriptors.

The steps in the bottom up analysis are given below. An explanation of the steps follows the example.

```
1.  i[∝]=n[25]*(i[γ]+i[β])      u*
2.       =n[25]*(i[γ]+i[β])     u[i[∝]▲=a]|u*
3.        n[25]*(i[γ]+i[β])     u[i[∝]=▲a]|u*
4.              *(i[γ]+i[β])    f[n[25]]|u[...]|u*
5.    f[n[25]]*(i[γ]+i[β])      u[i[β]=▲a]|u*
6.              *(i[γ]+i[β])    t[f[n[25]]]|u[...]|u*
7. t[f[n[25]]]*(i[γ]+i[β])      u[i[∝]=▲a]|u*
8.              *(i[γ]+i[β])    t[t[f[n[25]]]▲*f]|u[...]|u*
9.               (i[γ]+i[β])    t[t[f[n[25]]]*▲f]|u[...]|u*
10.               i[γ]+i[β])    f[(▲a)]|t[...]|u[...]|u*
11.                  +i[β])     u[i[γ]▲=a]|f[...]|t[...]|u[...]|u*
12.                     FAIL
13.                  +i[β])     a[i[γ]▲..a]|f[...]|t[...]|u[...]|u*
14.                     FAIL
15.                  +i[β])     f[i[γ]]|f[...]|t[...]|u[...]|u*
16.           f[i[γ]]+i[β])     f[(▲a)]|t[...]|u[...]|u*
17.                  +i[β])     t[f[i[γ]]]|f[...]|t[...]|u[...]|u*
18.       t[f[i[γ]]]+i[β])      f[(▲a)]|t[...]|u[...]|u*
19.                  +i[β])     t[t[f[i[γ]]]▲*f]|f[...]|t[...]|u[...]|u*
20.                     FAIL
21.                  +i[β])     a[t[f[i[γ]]]]|f[...]|t[...]|u[...]|u*
22.    a[t[f[i[γ]]]]+i[β])      f[(▲a)]|t[...]|u[...]|u*
23.                  +i[β])     a[a[t[f[i[γ]]]]▲+t]|f[...]|t[...]|u[...]|u*
24.                   i[β])     a[a[t[f[i[γ]]]]+▲t]|f[...]|t[...]|u[...]|u*
25.                      )      u[i[β]▲=a]|a[...]|f[...]|t[...]|u[...]|u*
26.                     FAIL
```

```
27.                )      s[i[β]▲..s]|a[...]|f[...]|t[...]|u[...]|u*
28.                FAIL
29.                )      f[i[β]]|a[...]|f[...]|t[...]|u[...]|u*
30.        f[i[β]])       a[a[t[f[i[γ]]]]+▲t]|f[...]|t[...]|u[...]|u*
31.                )      t[f[i[β]]]|a[...]|f[...]|t[...]|u[...]|u*
32.    t[f[i[β]]])        a[a[t[f[i[γ]]]]+▲t]|f[...]|t[...]|u[...]|u*
33.                )      t[t[f[i[β]]]▲*r]|a[...]|f[...]|t[...]|u[...]|u*
34.                FAIL
35.                )      a[t[f[i[β]]]]|a[...]|f[...]|t[...]|u[...]|u*
36. a[t[f[i[β]]]])        a[a[t[f[i[γ]]]]+▲t]|f[...]|t[...]|u[...]|u*
37.                APPLY 19,20; FAIL BOTH; RETURN TO STATE AT STEP 32.
38.    t[f[i[β]]])        a[a[t[f[i[γ]]]]+▲t]|f[...]|t[...]|u[...]|u*
39.                )      a[a[t[f[i[γ]]]]+t[f[i[β]]]]|f[...]|t[...]|u[...]|u*
40. a[a[t[f[i[γ]]]]+t[f[i[β]]]])
                           f[(▲a)]|t[...]|u[...]|u*
41.                APPLY 19,20; FAIL BOTH; RETURN TO STATE AT STEP 40.
42.                )      f[(a[a[t[f[i[γ]]]]+t[f[i[β]]]]▲)]|t[...]|u[...]|u*
43.          empty        f[(a[a[t[f[i[γ]]]]+t[f[i[β]]]])]|t[...]|u[...]|u*
44. f[(a[a[t[f[i[γ]]]]+t[f[i[β]]]])]
                           t[t[f[n[25]]]*▲r]|u[...]|u*
45.          empty        t[t[f[n[25]]]*f[(a[a[t[f[i[γ]]]]+t[f[i[β]]]])]]|u[...]|u*
46. t[t[f[n[25]]]*f[(a[a[t[f[i[γ]]]]+t[f[i[β]]]])]]
                           u[i[α]=▲a]|u*
47.                APPLY 17; FAIL;
48.          empty        a[t[f[n[25]]]*f[(a[a[t[f[i[γ]]]]+t[f[i[β]]]])]]|
                                u[...]|u*
```

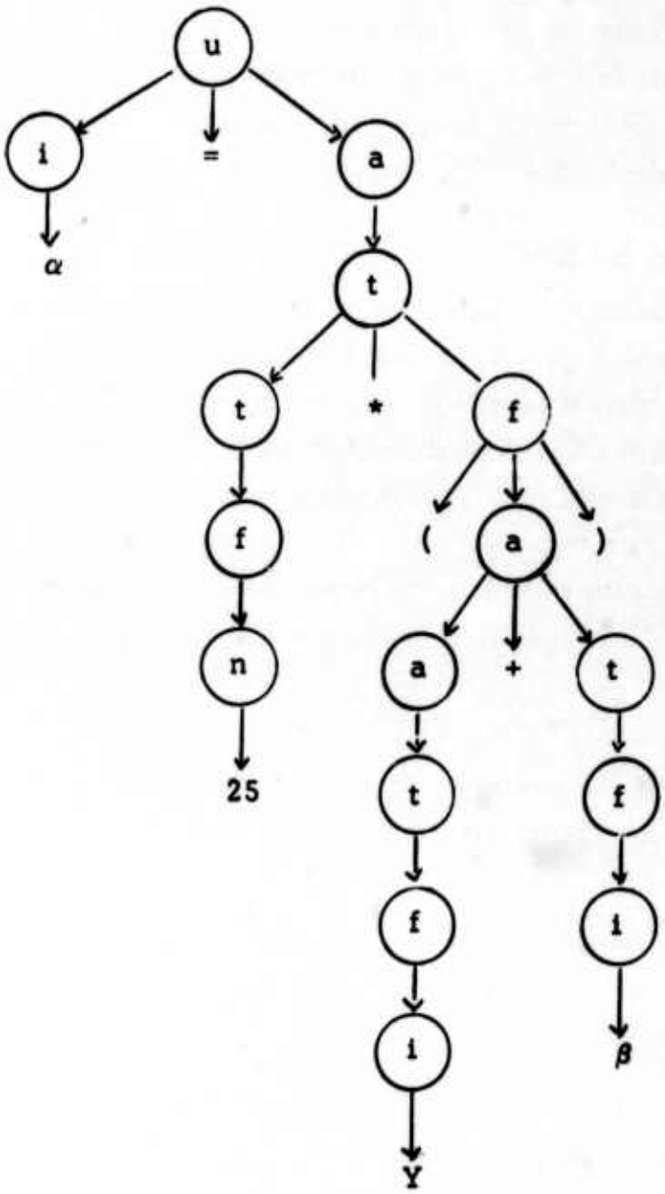
```
49.  a[t[f[n[25]]]*f[(a[a[t[f[i[Y]]]]+t[f[i[β]]]])]]
                                    u[i[α]=▲a]|u*

50.                  empty          u[i[α]=a[t[f[n[25]]]*f[(a[a[t[f[i[Y]]]]+
                                                  t[f[i[β]]]])]]|u*

51.                    FINISHED
```

2-33

Some remarks may help:

Step	
1	The final goal is u ("final" denoted by *); the input stream is i[α] = n[25] * . . .
2	Rule 11 indicates that the i[α] can be used to make a u; specifically we need to find an "=" followed by an a.
3	The "=" is found (it can "lead" to nothing else -- i.e., there is no rule for = → anything); move the pointer ▲ around the = and take it off the input list.
4	Rule 14 indicates that the n[25] can be used to make an f.
5	The f[n[25]] is completely recognized; place it on the front of the input list.
6	Rule 16 indicates that the f[n[25]] can be used to make a t.
7	The t[f[n[25]]] is completely recognized; place it on the front of the input list.
12	Failure because we need (the terminal) "=" and have a "+";
13	The i[Y] can also be used to make an s by rule 12.
	Etc.

At the end we have completed the tree:


u
i
=
a
α
t
t
*
f
f
(
a
)
n
a
+
t
25
t
f
f
i
i
β
γ

2.3.2. Bounded Context Analysis

Roughly speaking, bounded context analysis can be characterized as follows: at each point in the analysis, the decision as to what action to take next is a function of the symbol (character, descriptor) under scan and of N symbols on either side (where N is fixed for a given language).

2.3.2.1. Precedence Analysis

The first variation we shall discuss is that of precedence analysis, applicable to languages which qualify as precedence languages. In very rough terms, a language is a precedence language if no syntactic type has a definition which admits two defined types to occur without at least one terminal symbol between them (unless one of the defined types has a definition of the form <d>::= $t_1 \mid t_2 \mid \ldots \mid t_m$ where the t_j are terminal symbols) and, further, is such that between any pair (t_1, t_2) of terminal symbols no more than one of the relations: T_1 takes precedence over T_2 ($T_1 \gtrdot T_2$); T_1 yields precedence to T_2 ($T_1 \lessdot T_2$); or T_1 is equal in precedence to T_2 ($T_1 \doteq T_2$), holds; if no precedence relation is possible we denote this by "err".

For reasons discussed in an appendix*, language L_T is a precedence language and the "precedence matrix" for L_T is the following:

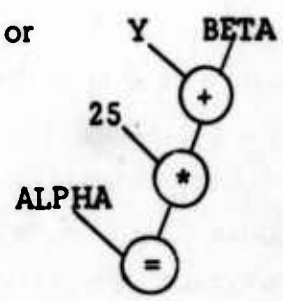
2-36

	<rlop>	<adop>	(	)	*	THEN	ELSE	=	TO	.	..
<rlop>	err	$\lessdot$	$\lessdot$	err	$\lessdot$	$\gtrdot$	err	err	err	err	err
<adop>	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$	err	err	$\gtrdot$	err
(	err	$\lessdot$	$\lessdot$	$\doteq$	$\lessdot$	err	err	err	err	err	err
*	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	err	err	$\gtrdot$	err
THEN	err	err	err	err	err	err	$\doteq$	$\lessdot$	$\lessdot$	err	err
ELSE	err	err	err	err	err	err	err	$\lessdot$	$\lessdot$	err	err
=	err	$\lessdot$	$\lessdot$	err	$\lessdot$	err	$\gtrdot$	err	err	$\gtrdot$	err
TO	err	err	err	err	err	err	$\gtrdot$	err	err	$\gtrdot$	err
.	err	err	err	err	err	err	err	$\lessdot$	err	$\doteq$	err
..	err	err	err	err	err	err	err	$\gtrdot$	$\gtrdot$	err	err

* The Floyd paper, "Syntactic Analysis and Operator Precedence".

If the phrases were output and numbered, taking some liberties with notation and rearrangement, we could have:

Line	Phrase Output			
1	ADD (Y, BETA)	or	$\mathfrak{D}_{ADD}\mathfrak{D}_{Y}\mathfrak{D}_{BETA}$	or
2	TIMES (25, (1))		$\mathfrak{D}_{TIMES}\mathfrak{D}_{25}\mathfrak{D}_{(1)}$	
3	STORE (ALPHA, (2))		$\mathfrak{D}_{STORE}\mathfrak{D}_{ALPHA}\mathfrak{D}_{(2)}$	



We will not delve further into this at the moment.

2-39

2.3.2.2 Production Analysis

An extremely interesting (and, as we shall see below, "best" method in our opinion) is a technique first suggested by R. W. Floyd and first implemented in a "practical" fashion by A. J. Perlis and associates. The original Floyd paper on the subject is included in the appendices ("A Descriptive Language for Symbol Manipulation").

The method can be thought of as a variation on precedence analysis wherein the flow of the analysis is not automatically provided by the precedence relations but is specified by the "programmer" for each "syntactic situation".

Suppose that we have a "descriptor stack" and a "descriptor register" as with precedence analysis. Suppose that we further have the following "actions" which can be specified.

1. Scan - causes the descriptor register contents to be placed on the top of the descriptor stack and a new descriptor fetched from the input stream into the descriptor register.

Syntactic analysis by the precedence technique is accomplished as follows: There is a descriptor "stack" which initially contains the descriptor for some (imagined) terminal symbol which yields precedence to all other terminal symbols. Descriptors are read from the input stream and placed into a "descriptor register". When any descriptor for a non-terminal symbol is read into the descriptor register, it is immediately placed on the top of the stack. When any descriptor for a terminal symbol is read into the descriptor register, the precedence relation between it and the most recent terminal (descriptor) on the stack is compared; if the "most recent" has greater precedence, then a "phrase" has been found, and it is replaced by a single "phrase descriptor". Otherwise the descriptor is moved from the descriptor register to the top of the stack and input is continued. The following example may give the flavor:

Descriptor stream: $\mathcal{D}_{\text{ALPHA}}\ \mathcal{D}_{=}\ \mathcal{D}_{25}\ \mathcal{D}_{*}\ \mathcal{D}_{(}\ \mathcal{D}_{Y}\ \mathcal{D}_{+}\ \mathcal{D}_{\text{BETA}}\ \mathcal{D}_{)}\ \mathcal{D}_{.}$

2-37

Step	Stack	Descriptor Register	Remarks
1	$\mathcal{D}_{\vdash}$	$\mathcal{D}_{\text{ALPHA}}$	ALPHA non-terminal; move to stack and continue
2	$\mathcal{D}_{\vdash}$ $\mathcal{D}_{\text{ALPHA}}$	$\mathcal{D}_{=}$	$\vdash \lessdot =$; move to stack and continue
3	$\mathcal{D}_{\vdash}$ $\mathcal{D}_{\text{ALPHA}}$ $\mathcal{D}_{=}$	$\mathcal{D}_{25}$	25 non-terminal; move to stack and continue
4	$\mathcal{D}_{\vdash}$ $\mathcal{D}_{\text{ALPHA}}$ $\mathcal{D}_{=}$ $\mathcal{D}_{25}$	$\mathcal{D}_{*}$	$= \lessdot *$; move to stack and continue
5	$\mathcal{D}_{\vdash}$ $\mathcal{D}_{\text{ALPHA}}$ $\mathcal{D}_{=}$ $\mathcal{D}_{25}$ $\mathcal{D}_{*}$	$\mathcal{D}_{(}$	$* \lessdot ($; move to stack and continue
6	$\mathcal{D}_{\vdash}$ $\mathcal{D}_{\text{ALPHA}}$ $\mathcal{D}_{=}$ $\mathcal{D}_{25}$ $\mathcal{D}_{*}$ $\mathcal{D}_{(}$	$\mathcal{D}_{Y}$	Y non-terminal; move to stack and continue
7	$\mathcal{D}_{\vdash}$ $\mathcal{D}_{\text{ALPHA}}$ $\mathcal{D}_{=}$ $\mathcal{D}_{25}$ $\mathcal{D}_{*}$ $\mathcal{D}_{(}$ $\mathcal{D}_{Y}$	$\mathcal{D}_{+}$	$(\lessdot +$; move to stack and continue
8	$\mathcal{D}_{\vdash}$ $\mathcal{D}_{\text{ALPHA}}$ $\mathcal{D}_{=}$ $\mathcal{D}_{25}$ $\mathcal{D}_{*}$ $\mathcal{D}_{(}$ $\mathcal{D}_{Y}$ $\mathcal{D}_{+}$	$\mathcal{D}_{\text{BETA}}$	BETA non-terminal; move to stack and continue
9	$\mathcal{D}_{\vdash}$ $\mathcal{D}_{\text{ALPHA}}$ $\mathcal{D}_{=}$ $\mathcal{D}_{25}$ $\mathcal{D}_{*}$ $\mathcal{D}_{(}$ $\mathcal{D}_{Y}$ $\mathcal{D}_{+}$ $\mathcal{D}_{\text{BETA}}$	$\mathcal{D}_{)}$	$+ \gtrdot)$; the first phrase has been found; since $(\lessdot +$ the phrase is $\mathcal{D}_{Y}$ $\mathcal{D}_{+}$ $\mathcal{D}_{\text{BETA}}$; remove and continue
10	$\mathcal{D}_{\vdash}$ $\mathcal{D}_{\text{ALPHA}}$ $\mathcal{D}_{=}$ $\mathcal{D}_{25}$ $\mathcal{D}_{*}$ $\mathcal{D}_{(}$ $\mathcal{D}_{\text{Phrase-1}}$ $\mathcal{D}_{)}$		$(\doteq)$; move to stack and continue
11	$\mathcal{D}_{\vdash}$ $\mathcal{D}_{\text{ALPHA}}$ $\mathcal{D}_{=}$ $\mathcal{D}_{25}$ $\mathcal{D}_{*}$ $\mathcal{D}_{(}$ $\mathcal{D}_{\text{Phrase-1}}$ $\mathcal{D}_{)}$	$\mathcal{D}_{.}$	$) \gtrdot .$; the second phrase has been found; since $(\doteq)$ and $* \lessdot ($ the phrase is $\mathcal{D}_{(}\mathcal{D}_{\text{Phrase-1}}\mathcal{D}_{)}$; replace and continue
12	$\mathcal{D}_{\vdash}$ $\mathcal{D}_{\text{ALPHA}}$ $\mathcal{D}_{=}$ $\mathcal{D}_{25}$ $\mathcal{D}_{*}$ $\mathcal{D}_{\text{Phrase-2}}$	$\mathcal{D}_{.}$	$* \gtrdot .$; the third phrase has been found; since $= \lessdot *$ the phrase is $\mathcal{D}_{25}$ $\mathcal{D}_{*}$ $\mathcal{D}_{\text{Phrase-2}}$; replace & continue
13	$\mathcal{D}_{\vdash}$ $\mathcal{D}_{\text{ALPHA}}$ $\mathcal{D}_{=}$ $\mathcal{D}_{\text{Phrase-3}}$	$\mathcal{D}_{.}$	$= \gtrdot .$; the fourth phrase has been found; since $\vdash \lessdot =$ the phrase is $\mathcal{D}_{\text{ALPHA}}\mathcal{D}_{=}\mathcal{D}_{\text{Phrase-3}}$; replace & continue
14	$\mathcal{D}_{\vdash}$ $\mathcal{D}_{\text{Phrase-4}}$	$\mathcal{D}_{.}$	

2. Phrase (M) causes the top n elements of the descriptor stack to be "output" as the "next" phrase and a descriptor of this phrase to replace these n descriptors on the stack.

3. Error - causes an error indication.

Suppose we further have a mechanism (recognizer) for detecting the presence of a specific descriptor in a specific position of the stack or descriptor register. The symbol "$\mathcal{R}$" will be used to denote the recognizer for identifier, literal, or phrase; the L_T terminal symbols themselves plus the symbols $\mathcal{V}$ and $\mathcal{I}$ will be used to denote their recognizers. The following (a "program" for assignment statements only) describes the method:

Rule	Stack	Descriptor Register	Action	Next Rule
1		$\mathcal{V}$	Scan	3
2			Error	
3		=	Scan	5
4			Error	
5		$\mathcal{R}$	Scan	8
6		(	Scan	5
7			Error	
8	$\mathcal{R} * \mathcal{R}$		Phrase(3)	8
9	$\mathcal{R} + \mathcal{R}$	*	Scan	5
10	$\mathcal{R} - \mathcal{R}$	*	Scan	5
11	$\mathcal{R} + \mathcal{R}$		Phrase(3)	8
12	$\mathcal{R} - \mathcal{R}$		Phrase(3)	8
13	$\mathcal{R} = \mathcal{R}$	.	Phrase(3)	done
14		)	Scan	17
15	$\mathcal{R}$		Scan	5
16			Error	
17	$(\mathcal{R})$		Phrase(3)	8
18			Error	

2-40

2-41

The analysis commences with the application of rule 1; failure to match the stack and descriptor register specified by a rule causes the next sequential rule to be applied. Success causes the action indicated to be performed and the specified rule to be tried next. Thus, in processing our old friend

$$\mathscr{D}_{\text{ALPHA}}\ \mathscr{D}_{=}\ \mathscr{D}_{25}\ \mathscr{D}_{*}\ \mathscr{D}_{(}\ \mathscr{D}_{Y}\ \mathscr{D}_{\text{BETA}}\ \mathscr{D}_{)}\ .$$

we would have the following trace:

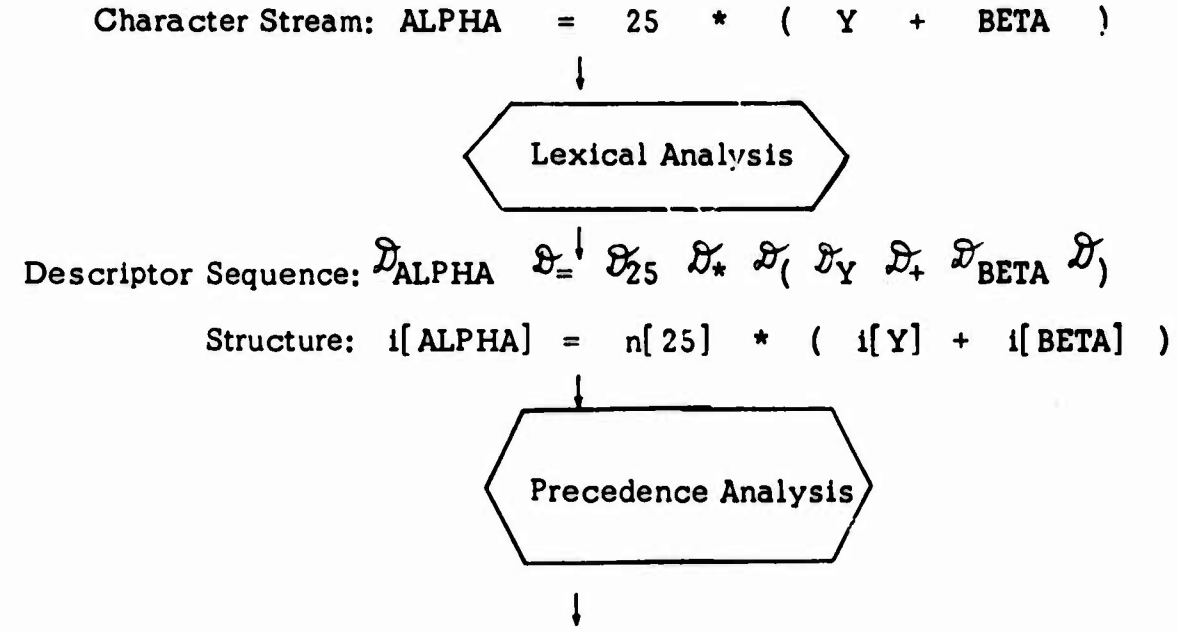
Step	Rule	Result
1	1	Scan
2	3	Scan
3	5	Scan
4	15	Scan
5	6	Scan
6	5	Scan
7	15	Scan
8	5	Scan
9	11	Phrase-1: $\mathscr{D}_{Y}\ \mathscr{D}_{+}\ \mathscr{D}_{\beta}$
10	14	Scan
11	17	Phrase-2: $\mathscr{D}_{(}\ \mathscr{D}_{\text{Phrase-1}}\ \mathscr{D}_{)}$
12	8	Phrase-3: $\mathscr{D}_{25}\ \mathscr{D}_{*}\ \mathscr{D}_{\text{Phrase-2}}$
13	13	Phrase-4: $\mathscr{D}_{\alpha}\ \mathscr{D}_{=}\ \mathscr{D}_{\text{Phrase-3}}$
14	done	

We will come back to this method later in section 3.

2.3.3. Mixed Type Analysis

It is possible to combine precedence analysis and bottom up predictive analysis. Note that in our example of bottom up analysis, we "fed" the analysis with descriptors of "partially analyzed" material; that is, we presumed that the lexical analysis had supplied the "i" and "n" structure brackets to the identifiers and integers. We can generalize this as follows: Suppose that a precedence analyzer is available and that it can analyze arithmetic expressions but nothing "larger". That is, it has a precedence matrix with entries $\lessdot$, $\gtrdot$, $\doteq$, err and "exit". Further, suppose that a syntactic type can be "attached" to each phrase by the precedence analyzer. Then, we can view the analysis as a precedence analysis on the sequence of descriptors produced by lexical analysis resulting in replacement of certain sub-sequence by single descriptors (phrases) followed by a bottom up predictive analysis on this reduced sequence.

Thus, we could have (considering our previous example again):



2-42

Reduced Descriptor Sequence: $\mathcal{D}_{ALPHA}\ \mathcal{D}_{=}\ \mathcal{D}_{Phrase-3}$

Structure: i[ALPHA] = a[Phrase-3]

↓

Predictive Analysis

↓

Structure: u[i[ALPHA] = a[Phrase-3]]

2-43

2.3.4. Miscellaneous Other Methods

Most of the methods currently used in compilers for performing syntactic analysis are very close to the three basic methods described above.

The last year or so has seen a host of "syntax directed" and "table driven" compilers being constructed and, in many instances, advertised as the newest, best, hottest, greatest, etc. compiler to ever make the scene. Upon close inspection, however, it is seen that almost all of these are minor variations on the various themes developed above, often sufficiently ill described that the relationship is not especially evident without considerable study. There are, however, two other methods which vary enough from these to comment upon.

A variation on predictive analysis has been devised by M. E. Conway and is reported in: Conway, M. E., "Design of a Separable Transition Diagram Compiler", Communications of the ACM, Vol. 6, pp 396-408, July, 1963.

Conway has applied this method to COBOL and indicates that it can also be used to handle analysis of "scientific" languages. By using the technique he describes he was able to produce a rather fast, quite elegant, COBOL compiler very rapidly.

2-44

Another technique that justifies mention is that used in many (IBM produced) FORTRAN-II compilers. Basically, they perform several scans over the source text performing a lexical analysis and then doing a complete parenthesization of arithmetic expressions and giving a "level" code to each sub-computation. They use "level" to encode both "depth within parentheses" as well as "precedence of operators". Given a list of computations with appropriated sequence numbering and level coding they can then "sort out" different kinds of sub-computations for special consideration, including certain kinds of common sub-computation elimination.

2.4 Error Analysis and Recovery

In a practical compiler, one of the major considerations in the choice of analysis technique is the ability to detect and recover from errors in source programs. The syntactic analysis techniques sketched above vary considerably in their properties with respect to error analysis and recovery.

In "top down" predictive analysis, for example, an error is detected (for example in Syntax II) when the analyzer cannot recognize a <prog>. Although the exact point in the input string past which recognition fails will be known, it is extremely difficult to determine exactly why the error occurred and to, in a general way, devise means for recovery. Further, if the error is at the "end" of some construction, all the correct input will have been "forgetten".

Several schemes exist for dealing with this problem, notably:

1. A scheme which permits specification of "no back up" on certain constructs. For example, in Syntax II, no back up on recognition of "=" or "(" could help isolate the reasons for a failure.

2. A scheme due to E. T. Irons* which, in effect, carries along all possible parses of an input string.

3. Special "error" syntactic types which could be defined in the syntax. For example, one might define <factor> by

<factor>:: <ident> | <integer> | (<ae>) | (<ae> <null>

to allow missing right parentheses to be neglected. The ramifications of this are not trivial, however.

At the present time there is no completely satisfactory general scheme for dealing with syntactic errors discovered in the course of top down predictive analysis.

In the case of bottom up predictive analysis, the situation is very similar to that of the top down method. However, there is an advantage with bottom up analysis in that when an error is detected, the symbols in the input string which correctly "fit" some interpretation will not have been "forgotten". To be a bit less vague, if a top down analysis (considering Syntax II) is given the goal of <ustat> and inspects the string

X = Y + Z*(A+B* (C+D).

it will not be able to parse the string because of the missing right parenthesis. Indeed, it will completely analyze the string up to the . and then completely "unwind" the analysis ending up with an error report and the "next" input string

---

* Irons, E. T., "A Syntax Directed Compiler for ALGOL-60", Comm. ACM 4 (1961), 51-55

symbol being X.

Bottom up analysis, on the other hand, will similarly analyze up to the . but will fail with ")" missing and "no place to go" but will be pointing at the . rather than "unwinding" to the X.

Bounded context analysis, on the other hand, is much better suited to error recovery procedures. Thus, given the above input, the precedence analyzer would be in the following state:

Stack: $\vartheta \vdash \vartheta_X \vartheta = \vartheta_Y \vartheta + \vartheta_Z \vartheta * \vartheta (\vartheta_{\text{Phrase-3}}$

Descriptor Register: $\vartheta .$

Phrase List: Phrase-1: $\vartheta_C \vartheta + \vartheta_D$

Phrase-2: $\vartheta (\vartheta_{\text{Phrase-1}} \vartheta)$

Phrase-3: $\vartheta_A \vartheta + \vartheta_{\text{Phrase-2}}$

and be stopped because there is no precedence relation between the pair "(" and ".". It is quite easy to paste in recovery mechanisms on such error pairs. The situation is similar with productions analysis. Indeed, since a production analysis is, in effect, specified by a program, the extension of the program to include error analysis and recovery are a relatively straightforward process. It is this property which, in our opinion, makes productions analysis preferable over any of the other bounded context methods. In the productions analysis program given above, steps 2, 4, 7, 16, and 18 are "error steps"; the kind of error at each is as follows:

2-46

2: The first symbol in the assignment statement is not an identifier.

4: The first identifier ("left hand side") is not followed by an =.

7: An arithmetic expression starts with something which is not an identifier, integer or phrase ($\mathcal{R}$) or a left parenthesis.

16: The quantity at the top of the stack does not combine with any quantities to its' left in an allowable fashion (rules 8 - 13), is not followed by a right parenthesis (rule 14), and, indeed, is not a quantity (rule 15).

The "missing right parenthesis" recovery would be pasted in here; e.g.,

Rule	Stack	Descriptor Register	Action	Next Rule
15A	(		ADD())	15B
15B			Phrase(3)	8

Here Add (σ) causes symbol (descriptor) σ to be placed on top of stack.

18: A quantity followed by a right parenthesis is lacking the balancing left parenthesis (i.e., we get to rule 17 via success rule 14 only.)

2-47

2.5 Mechanical Generation of Syntactic Analyzers from Syntax Specifications

It is clear that any reasonable implementation of the analyzer algorithms described above would result in a "table driven" program. Indeed, a general purpose top down analyzer which accessed a (more-or-less direct) encoding of the syntax specification is rather simple to envision. (That is to say that the idea of somehow encoding a particular syntax specification algorithmically would probably not occur to anyone trying to implement such an analyzer.) A bottom up analyzer can also, in a reasonably natural way, access an encoded version of the syntax specification directly. In this case, however, the syntax specification has to be "twisted around" a bit. That is, bottom up analysis requires an encodement which allows going from some syntactic type in hand to all the possible things which can be built from it. The specification discussed in section 2.3.1.2 is in a form suitable for driving a general purpose bottom up analyzer.

The precedence analyzer is also driven by a data set, namely the precedence matrix. It is not so obvious that this matrix can be derived mechanically from a syntax specification. However, a recent result due to R. W. Floyd* shows that such a mechanical procedure does exist

We have, as noted, thought of a productions analysis specification as a program. Again, any implementation of a productions analyzer would most likely take the form of a table driven device. Some recent work by Earley and others at Carnegie Tech indicates that it is possible to mechanically produce (the parse part of) a productions analysis program from a proffered syntax in many cases. Since much of the interpretation of the parse must be tailored to the further processing to be done (like code selection) and thus cannot be mechanically generated (unless, perhaps, the desired result were polish pre-(post-) fix or some similar form) it is not terribly important whether or not the

---

* See Floyd, "Syntactic Analysis and Operator Precedence", in the appendices.

productions for parsing are automatically prepared. In practice most people can learn to use productions analysis with facility quite rapidly.

3 INTERPRETING THE PARSE

3.0 General Discussion

Parsing, as we have described the process above, results in either a syntax tree depicting the complete syntactic analysis of the input or a series of phrases which can be thought of as a somewhat less complete syntax tree. The "interpretation" of the parse, as we shall understand the term in these notes consists of three basic kinds of activity:

1. Generation of some sort of pseudo-code representation of the sequence of computation from the tree or phrases resulting from the syntactic analysis.
2. Handling the declarations discovered in the parsing.
3. Adjusting the sequence of computations with respect to type conversion, scaling, expansion of certain implicit functions, and generally getting the sequence of computations into a shape which will allow the collection of information pertinent to optimization and, eventually, the generation of machine code.

If the language being translated and the machine* for which we are generating code are sufficiently well behaved, the interpretation of the parse can result in the generation of actual machine coding rather than pseudo code. However, for our present purposes we will not consider the direct generation of machine code from the parse. (See the Cheatham and Sattley paper in the appendices.)

---

* We mean "machine" in the sense of both the hardware and the software interface between the compiler and the hardware -- sophisticated assembly routines, for example.

3.1 Generation of Pseudo-Code

Let us identify the problem here by continuing the examples presented in section 2.0 through the generation of pseudo code. For the present, we think of the following simple pseudo instructions*:

	Instruction	Arguments	Interpretation
1)	PLUS	X Y	X + Y
2)	MINUS	X Y	X - Y
3)	TIMES	X Y	X * Y
4)	OVER	X Y	X + Y
5)	STORE	X Y	X ← Y
6)	GOTO	L	transfer control to (label) L

Let us think of the arguments of these instructions as being descriptors in the sense defined above -- i.e., table code and line within table of a table housekeeping the representation of the value. Indeed, let us further presume an "instruction" table which contains (at least) entries for the six pseudo-instructions defined above. Then, a pseudo-instruction is a pair or triple, as for example:

$$\mathcal{D}_{\text{PLUS}}\ \mathcal{D}_{\text{Y}}\ \mathcal{D}_{\text{BETA}}$$

(Note: In the sequel we will often use the notations PLUS and $\mathcal{D}_{\text{PLUS}}$, + and $\mathcal{D}_{+}$, etc. interchangeably.)

If we further introduce the table in which we place the pseudo-instructions as a table for which we can have descriptors and utilize the notation $\mathcal{D}_{(i)}$ for the $i^{\underline{th}}$ line of that table, and further argue that a line of that table

---

* Note: Throughout these notes we use the terms instruction and operation interchangeably. They may refer to "pseudo" or "machine" instructions/operations also.

(let's call it the descriptor table) contains one descriptor, then we find that the computation

```
ALPHA = 25 * ( Y + BETA )
```

results in descriptor table entries as follows (assuming for no good reason that line 66 is the first one available to us when we start processing the above):

LINE	Descriptor Table Entry
⋮	⋮
66	D PLUS
67	D Y
68	D BETA
69	D TIMES
70	D 25
71	D (66)
72	D STORE
73	D ALPHA
74	D (69)
⋮	⋮

or, more readably, perhaps

LINE	Descriptors		
⋮			
66	PLUS	Y	BETA
69	TIMES	25	(66)
72	STORE	ALPHA	(69)
⋮		⋮	

Now, there are several kinds of manipulation of the material in this form which are rather naturally carried out following the syntactic analysis and generation of the pseudo code but preceding the optimization and machine

.code selection processing to come later.

These manipulations could include the following:

1. Type conversion (e.g., float an integer involved in floating computations)
2. Compile time computations (e.g., if 2 + 2 appears in the input, produce the 4 at compile time rather than at run time).
3. Scaling fixed point computations (i.e., type conversion of a more interesting and difficult type)
4. Code expansion (e.g., "open" code for generic functions; expansion of mapping or accessing functions -- the idea of "macros" of pseudo-coding)
5. Commutation of commutative operands to some canonical form.

2-53

We must, however, emphasize that we are not asserting that this is the place, in the compilation process, to do these kinds of manipulation. Indeed, our whole theme is that there are many ways and times to do the various manipulations required; the only ordering that we will assert as universal is that one should parse the input before he generates code. The time and place to do, for example, scaling and type conversion is dependent upon the rules in the programming language and upon the structure of the computer for which code is being generated. Thus, if one has a machine with an accumulator and a quotient register which can be connected for shifting, it may well be that scaling determination be deferred until code selection time. On the other hand, if one has a machine with no such accumulator-quotient arrangement, the determination of scaling might as well be done earlier -- even at parse time.

Let us suppose that by one means or another we have parsed the source language (or some interesting part of it, like a statement, do loop, block, or the like) and have produced a sequence of pseudo-operations in

the descriptor table. For example, referring back to the example of section 3.1 we have eaten up the string

ALPHA = 25 * (Y + BETA)

and emitted the sequence of descriptors.

Descriptor Table

Line	Descriptor
⋮	⋮
66	PLUS
67	Y
68	BETA
69	TIMES
70	25
71	(66)
72	STORE
73	ALPHA
74	(69)
⋮	⋮

Let us introduce a couple more pieces of trickery and we're ready to generate pseudo-code. First, to help accounting for who's who in the descriptor table, we add a one bit field to the descriptors which bit is 0, 1 as the descriptor is not, is the last argument of an instruction. Thus a symbol descriptor is now a triple, (TABLE, LINE, LAST) (where LAST ~ last argument). We will denote such "last" descriptors pictorially by the use of a super-script asterisk; thus, the example is

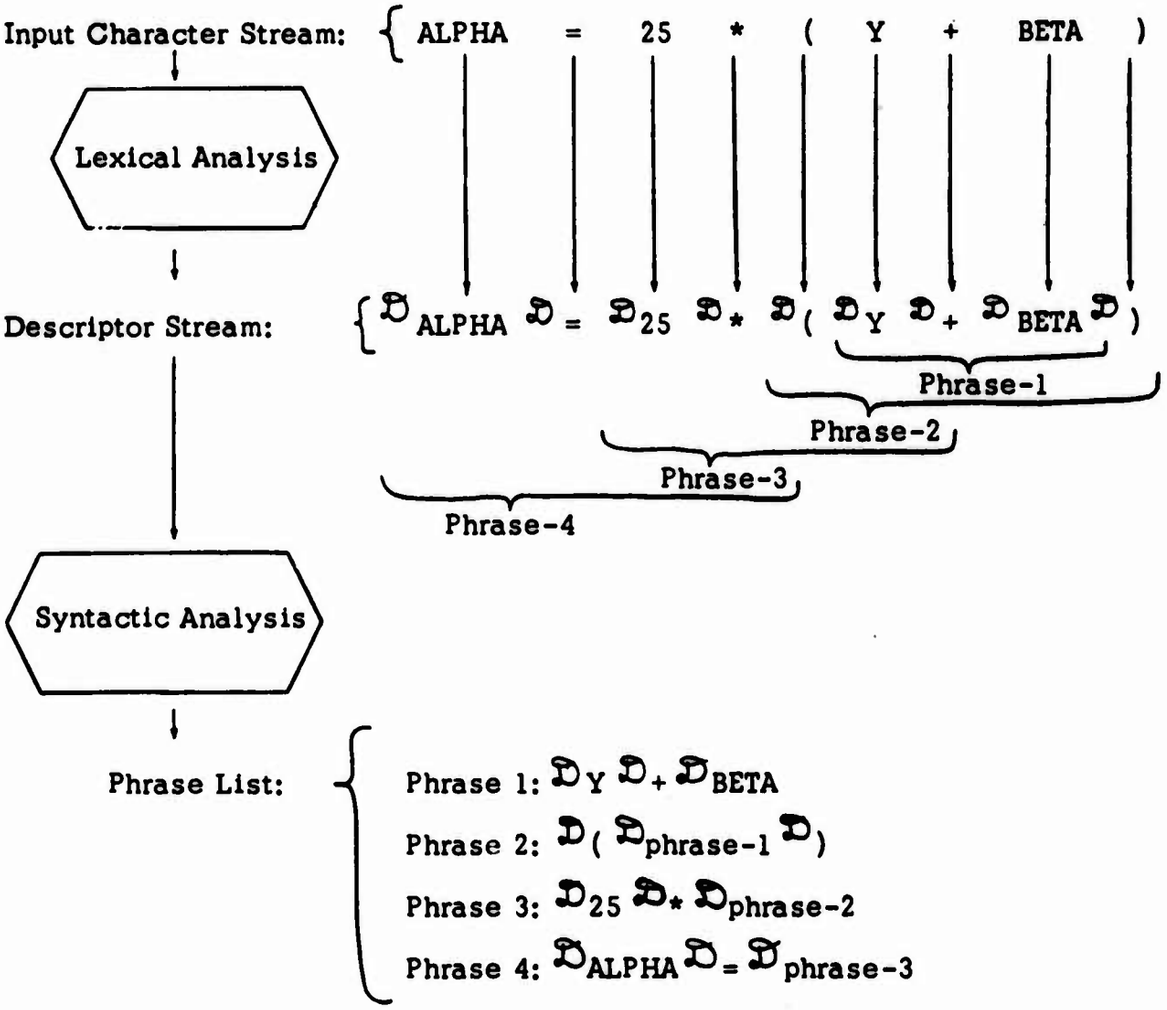
Line	Descriptor Table Entry
⋮	⋮
66	$\mathcal{D}_{\text{PLUS}}$
67	$\mathcal{D}_{\text{Y}}$
68	$\mathcal{D}^{*}_{\text{BETA}}$
69	$\mathcal{D}_{\text{TIMES}}$
70	$\mathcal{D}_{25}$
71	$\mathcal{D}^{*}_{(66)}$
72	$\mathcal{D}_{\text{STORE}}$
73	$\mathcal{D}_{\text{ALPHA}}$
74	$\mathcal{D}^{*}_{(69)}$

2-55

Finally, we introduce the function EMIT (....) which takes an arbitrary number of descriptor-valued arguments and produces a descriptor value (output). Specifically, EMIT (A_1, ... ,A_n) places the descriptors A_1,...,A_n in the "next" n lines in the descriptor table, marks the descriptor A_n as "last" and produces a descriptor of the form $\mathcal{D}_{(i)}$ where i is the line into which A_1 was placed.

Now let us reconsider the precedence analysis discussed in section 2.0.

Analysis of Source into a List of Phrases



2-56

From this it is fairly clear that the following "actions" will produce the desired sequence of pseudo-instructions from the phrase analysis of the proffered input:

Phrase-1 ← EMIT (D_{PLUS}, D_{Y}, D_{BETA})
Phrase-2 ← Phrase-1
Phrase-3 ← EMIT (D_{TIMES}, D_{25}, $D_{Phrase-2}$)
Phrase-4 ← EMIT (D_{STORE}, D_{ALPHA}, $D_{Phrase-3}$)

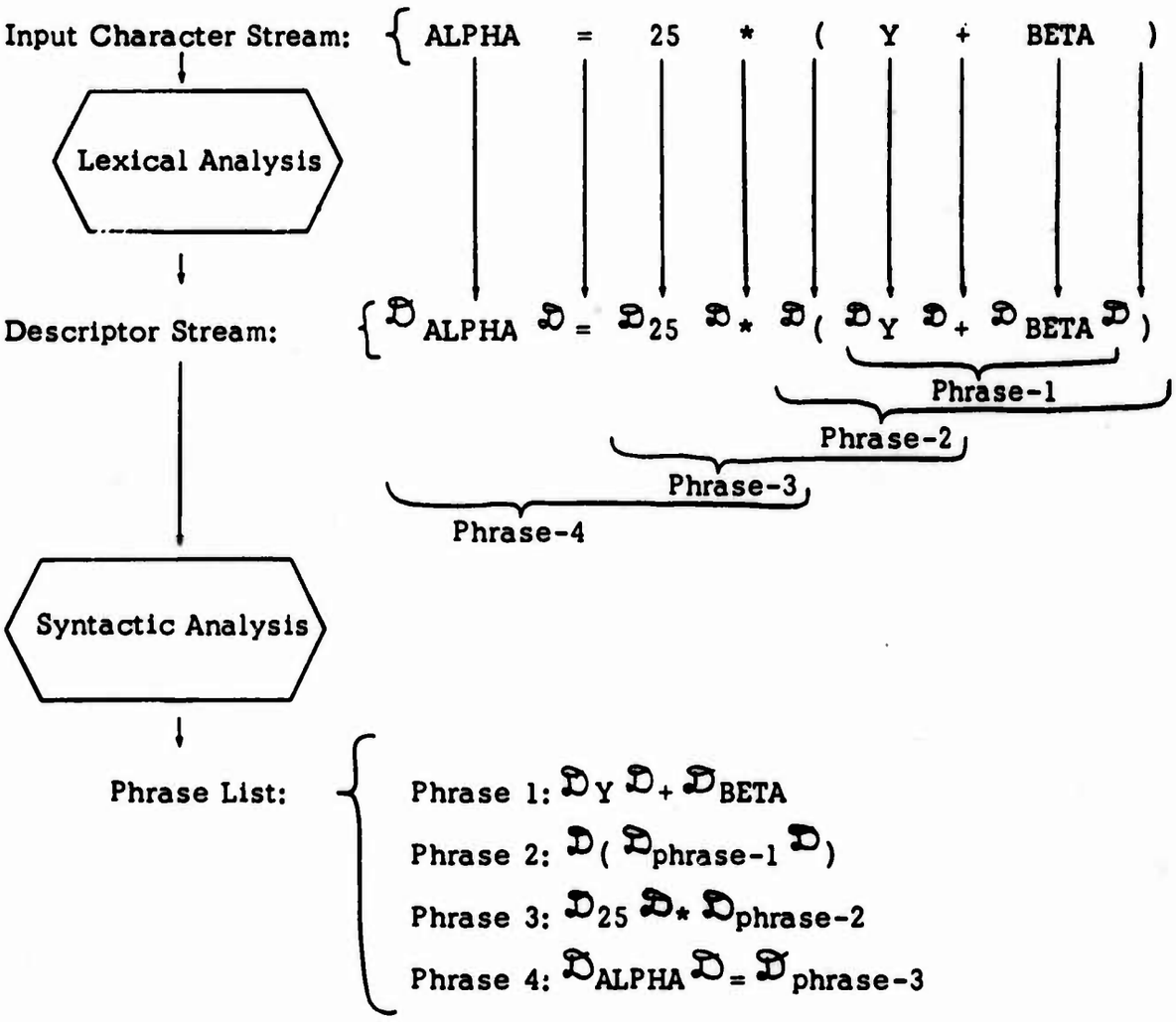
Line	Descriptor Table Entry
⋮	⋮
66	$\mathcal{D}$ PLUS
67	$\mathcal{D}$ Y
68	$\mathcal{D}^*$ BETA
69	$\mathcal{D}$ TIMES
70	$\mathcal{D}$ 25
71	$\mathcal{D}^*$ (66)
72	$\mathcal{D}$ STORE
73	$\mathcal{D}$ ALPHA
74	$\mathcal{D}^*$ (69)

2-55

Finally, we introduce the function EMIT (....) which takes an arbitrary number of descriptor-valued arguments and produces a descriptor value (output). Specifically, EMIT (A_1, ... ,A_n) places the descriptors A_1,...,A_n in the "next" n lines in the descriptor table, marks the descriptor A_n as "last" and produces a descriptor of the form $\mathcal{D}_{(i)}$ where i is the line into which A_1 was placed.

Now let us reconsider the precedence analysis discussed in section 2.0.

Analysis of Source into a List of Phrases



2-56

From this it is fairly clear that the following "actions" will produce the desired sequence of pseudo-instructions from the phrase analysis of the proffered input:

Phrase-1 ← EMIT (PLUS, Y, BETA)
Phrase-2 ← Phrase-1
Phrase-3 ← EMIT (TIMES, 25, Phrase-2)
Phrase-4 ← EMIT (STORE, ALPHA, Phrase-3)

Recall from section 2.3.2 that the precedence analysis technique operates by placing the descriptors from the descriptor stream into a "descriptor register". When a terminal symbol descriptor is placed in the descriptor register the precedence relation between it and the most recent terminal symbol in the stack is inspected, resulting either in the identification of a phrase or in moving the descriptor from the descriptor register to the top of the stack and obtaining of the next descriptor from the input stream and placing it in the descriptor register. The phrases isolated (at least in our simple example above) have one of the six forms:

X + Y
X - Y
X * Y
X / Y
X = Y
(X)

2-57

Where X, Y stand for arbitrary descriptors.

Now suppose we have an operator, PHRASE ( ) which takes a descriptor argument and performs the following function: If the top n positions of the stack contain a phrase, then PHRASE (A) causes the top n positions to be excised, and the argument, A, to be placed on top of the stack. Then, the interpretation of the phrase and generation of pseudo-instructions can be described as follows:

When a phrase has been isolated, perform the following, according to which pattern applies.

Pattern	Action
X + Y	PHRASE (EMIT (PLUS, X, Y))
X - Y	PHRASE (EMIT (MINUS, X, Y))
X * Y	PHRASE (EMIT (TIMES, X, Y))
X / Y	PHRASE (EMIT (OVER, X, Y))
X = Y	PHRASE (EMIT (STORE, X, Y))
(X)	PHRASE (X)

Now let us turn our attention to productions analysis again. We postulate the following:

1. The descriptors have four fields: TABLE, LINE, LAST, and TYPE. The first three are as above; the TYPE field will contain syntactic type (generally any "type" information useful) and be dealt with as described below.

2. There is defined a set of pattern elements (predicates) as follows (each string of characters above a wiggly line names a predicate):

a. $+, $-, $*, $TO, $=, etc. which are true, false as the descriptor to which they are applied is, is not that for the terminal character (string) "+", "-", "*", "TO", "=", etc.

b. IDENT, INTEGER which is true if the descriptor to which it is applied has as table code the code for the symbol table, literal table, respectively.

c. RLOP, ADOP which is true if the descriptor to which it is applied is a relation operator, add operator respectively.

d. Several special pattern elements, FACTOR, TERM, AE, REL, CSTAT, USTAT, STAT, PROG which are true if the TYPE field of the descriptor to which they are applied has the corresponding syntactic type codes (per, for example, the numbering of syntax I or II in section 2.1). The occurrence of these identifiers outside pattern elements will be taken to be equivalent to a literal whose value is the code for the particular syntactic type.

e. OTHERWISE - always true.

3. There is a stack of symbol descriptors and a "current register". Patterns can be applied to the top n positions of the stack and/or the current register. A pattern is true if all the pattern elements are true predicates when applied to the corresponding stack and/or current register contents. We will utilize the following notation for patterns (Π, Π_j name predicates or pattern elements):

$\ldots \Pi_n \Pi_{n-1} \ldots \Pi_0 \,//\, \rightarrow$ test the top (n+1) position of the stack.

$\ldots \Pi_n \ldots \Pi_0 \,//\, \Pi \,///\rightarrow$ test the top n+1 position of the stack and the current register.

$//\, \Pi \,///\rightarrow$ test the current register.

Patterns may optionally be labelled by placing an identifier in front of them.

4. We have a set of actions as follows:

a. SCAN(n) - lexically analyze for n descriptors placing the last one in the current register and placing the other n-1 plus the one initially in the current register into the stack.

b. TRY(ℓ) - starting with the pattern labelled ℓ, apply patterns until first one matches; then do the actions associated with it.

c. EMIT(d_1, d_2, ... ,d_n) - place the n descriptors $d_1, \ldots, d_n$ into the output list indicating d_n is "last"; take the descriptor of this "computation" as the result of the EMIT( ) function.

d. PHRASE(d) - excise all those elements of the stack involved in the most recent successful pattern and insert descriptor d on the stack.

e. Arithmetic (+, -, *, /) over literals, variables and fields of descriptors and substitution (=) of integer valued or descriptor valued variables.

2-59

2-58

f. CYCLE - nullifies the symbol in the current register.

g. ERROR - announces an error condition.

h. EXCISE - remove the n symbol descriptors on the stack involved in the most recent successful pattern from the stack.

i. TO(a) - perform the action labeled "a" next.

Actions are delimited by a period; an action may be labeled by placing an identifier followed by a double period in front of it.

5. We denote by COMP(n) the descriptor currently in the (top-n)$^{\underline{th}}$ position of the symbol descriptor stack and by TYPE(COMP(n)), the type field of COMP(n), etc.

6. Patterns are applied sequentially from the first in "tried" until a successful one is found; then the first action associated is performed and actions are performed sequentially until a control break is reached (i.e., a pattern is specified to be tried (TRY(p)) or a transfer of control in actions occurs (TO(a))).

The following schema provides a program for parsing optionally labeled unconditional statements. We have presumed pseudo instructions of above (PLUS, MINUS, TIMES, LABEL, and GOTO) plus the operation STOPPER with the obvious interpretation. It is presumed that, initially, there is a symbol descriptor in the current register and that pattern U0 is "tried".

2-60

```
 U0                          // IDENT   ///  SCAN(1). TRY(U01).
                             // $TO     ///  SCAN(2). TRY(T01).
                             // $STOP   ///  SCAN(1). TRY(STOP1).
     OTHERWISE                                ERROR.

 U01 ... IDENT               //  $=     ///  SCAN(1). TRY(AE0).
     ... IDENT               //  $..    ///  PHRASE(EMIT(LABEL,COMP(0)).
                                             EXCISE. CYCLE. SCAN(1).
                                             TRY(U0).
     OTHERWISE                                ERROR.

 AE0                         //  $(     ///  SCAN(1). TRY(AE0).
                             // IDENT   ///
                             // INTEGER ///  SCAN(1).
                                    AE0A..  TYPE(COMP(0))=FACTOR. TRY(F1).
     OTHERWISE                                ERROR.

  F1 ... TERM $* FACTOR//                     PHRASE(EMIT(TIMES,COMP(2),
                                                COMP(0))).
     ...          FACTOR//                    TYPE(COMP(0))=TERM. TRY(T1).
     OTHERWISE                                ERROR.

  T1 ...          TERM    //  $*     ///  SCAN(1). TRY(AE0).
     ... AE $+    TERM    //              PHRASE(EMIT(PLUS,COMP(2),
                                                COMP(0))).  TO(T1A).
     ... AE $-    TERM    //              PHRASE(EMIT(MINUS,COMP(2),
                                                COMP(0)).
     ...          TERM    //       T1A..  TYPE(COMP(0))=AE. TRY(AE1).
     OTHERWISE                                ERROR.

 AE1 ...             AE   // ADOP    ///  SCAN(1). TRY(AE0).
     ...          $( AE   //  $)     ///  PHRASE(COMP(0)).
                                             CYCLE.  SCAN(1).  TO(AE0A).
     ... IDENT $= AE     //              PHRASE(EMIT(STORE(COMP(2),
                                                COMP(0))).
                                    AE1A..  TYPE(COMP(0))=USTAT. TRY(U1).
     OTHERWISE                                ERROR.

 T 1 ...    $TO IDENT    //                  PHRASE(EMIT(GOTO,COMP(0)).
                                                TO(AE1A).
     OTHERWISE                                ERROR.

STOP1 ...          $STOP //                  PHRASE(EMIT(STOPPER)).
                                                TO(AE1A).
     OTHERWISE                                ERROR.
```

2-60

2-61

Analysis of the fragment

ALPHA = 25 * (Y + BETA) .

yields (where the pair indicates type and "quantity"):

```
  ⋮
66   PLUS     (AE, Y)          (TERM, BETA)
69   TIMES    (TERM, 25)       (FACTOR, (66))
72   STORE    (IDENT, ALPHA)   (AE, (69))
```

with the stack and current register state

... (USTAT, (72)) // $. ///

Supposing that we desired to produce a syntax tree rather than pseudo code, consider the following schema:

2-62

```
 U0                    // IDENT   ///  SCAN(1). TRY(U01).
                       // $TO     ///  SCAN(2). TRY(TO1).
                       // $STOP   ///  SCAN(1). TRY(STOP1).
    OTHERWISE                           ERROR.

 U01 ...       IDENT   //  $=     ///  SCAN(1). TRY(AE0).
     ...       IDENT   //  $..    ///  SCAN(1). TRY(U0).
    OTHERWISE                           ERROR.

 AE0                   //  $(     ///  SCAN(1). TRY(AE0).
                       // IDENT   ///
                       // INTEGER ///  SCAN(1).
                                       PHRASE(EMIT(COMP(0))).
                                AE0A.. TYPE(COMP(0))=FACTOR. TRY(F1).
    OTHERWISE                           ERROR.

  F1 ... TERM $* FACTOR//               PHRASE(EMIT(COMP(2),COMP(1),
                                          COMP(0))).
                                       TO(F1A).
     ...         FACTOR//               PHRASE(EMIT(COMP(0))).
                                 F1A.. TYPE(COMP(0))=TERM. TRY(T1).
    OTHERWISE                           ERROR.

  T1 ...         TERM  // $*     ///  SCAN(1). TRY(AE0).
     ... AE ADOP TERM  //               PHRASE(EMIT(COMP(2),COMP(1),
                                          COMP(0))).
                                       TO(T1A).
     ...         TERM  //               PHRASE(EMIT(COMP(0))).
                                 T1A.. TYPE(COMP(0))=AE. TRY(AE1).
    OTHERWISE                           ERROR.

 AE1 ...           AE  // ADOP   ///  SCAN(1). TRY(AE0).
     ...       $( AE   //  $)    ///  PHRASE(EMIT($(,COMP(0),$)).
                                          CYCLE. SCAN(1).
                                       TO(AE0A).
     ... IDENT $= AE   //               PHRASE(EMIT(COMP(2),COMP(1),
                                          COMP(0))).
                                AE1A.. TYPE(COMP(0))=USTAT. TRY(U1).
    OTHERWISE                           ERROR.

 TO1 ...   $TO IDENT   //               PHRASE(EMIT($TO,COMP(0)).
                                       TO(AE1A).
    OTHERWISE                           ERROR.

STOP1 ...      $STOP   //               PHRASE(EMIT(COMP(0)). TO(AE1A).
    OTHERWISE                           ERROR.

  U1 ...       USTAT   // $.    ///  PHRASE(EMIT(COMP(0),$.). CYCLE.
                                 U1A.. TYPE(COMP(0))=STAT. TRY(S1).
    OTHERWISE                           ERROR.
  S1 ... IDENT $.. STAT//               PHRASE(EMIT(COMP(2),COMP(1),
                                          COMP(0))).
                                       TO(U1A).
     ...      PROG STAT//               PHRASE(EMIT(COMP(1),COMP(0))).
                                       TO(S1A).
     ...           STAT//               PHRASE(EMIT(COMP(0))).
                                 S1A.. TYPE(COMP(0))=PROG. SCAN(1).
                                       TRY(U0).
    OTHERWISE                           ERROR.
```

2-62

2-63

With this scheme, the analysis of

ALPHA = 25 * (Y + BETA)

would yield the following (The pairs indicate syntactic type and "quantity"):

line	Stack and current register contents
⋮	
66 [INTEGER, 25]	... [IDENT,ALPHA] $= [FACTOR, (66)] // $* ///
67 [FACTOR, (66)]	... [IDENT,ALPHA] $= [TERM, (67)] // $* ///
68 [IDENT, Y]	... [IDENT,ALPHA] $= [TERM, (67)] $* $([FACTOR, (68)] // $+ ///
69 [FACTOR, (68)]	... [IDENT,ALPHA] $= [TERM, (67)] $* $([TERM, (69)] // $+ ///
70 [TERM, (69)]	... [IDENT,ALPHA] $= [TERM, (67)] $* $([AE, (70)] // $+ ///
71 [IDENT, BETA]	... [IDENT,ALPHA] $= [TERM, (67)] $* $([AE, (70)] $+ [FACTOR, (71)] // $) ///
72 [FACTOR, (71)]	... [IDENT,ALPHA] $= [TERM, (67)] $* $([AE, (70)] $+ [TERM, (72)] // $) ///
73 [AE, (70)] $+ [TERM, (72)]	... [IDENT,ALPHA] $= [TERM, (67)] $* $([AE, (73)] // $) ///
76 $([AE, (73)] $)	... [IDENT,ALPHA] $= [TERM, (67)] $* [FACTOR, (76)] // $. ///
79 [TERM, (67)] $* [FACTOR, (76)]	... [IDENT,ALPHA] $= [TERM, (79)] // $. ///
82 [TERM, (79)]	... [IDENT,ALPHA] $= [AE, (82)] // $. ///
83 [IDENT, ALPHA] $= [AE, (82)]	... [USTAT, (83)] // $. ///
86 [USTAT, (83)] $.	... [STAT, (84)] //

Consider, further, the following schema:

2-65

```
 U0                          // IDENT    ///  SCAN(1). TRY(U01).
                             // $TO      ///  SCAN(2). TRY(T01).
                             // $STOP    ///  SCAN(1). TRY(STOP1).
     OTHERWISE                                ERROR.

 U01 ...        IDENT        // $=       ///  SCAN(1). TRY(AE0).
     ...        IDENT        // $..      ///  PHRASE(EMIT(LABEL,COMP(0)).
                                              EXCISE. CYCLE. SCAN(1). TRY(U0).
     OTHERWISE               ERROR.

 AE0                         // $(       ///  SCAN(1). TRY(AE0).
                             // IDENT    ///
                             // INTEGER  ///  SCAN(1). TRY(F1).
     OTHERWISE                                ERROR.

  F1 ...      R $* R         //                PHRASE(EMIT(TIMES,COMP(2),
                                                COMP(0))). TRY(F1).
  T1 ...          R          // $*       ///  SCAN(1). TRY(AE0).
     ...      R $+ R         //                PHRASE(EMIT(PLUS,COMP(2),
                                                COMP(0))). TRY(T1).
     ...      R $- R         //                PHRASE(EMIT(MINUS,COMP(2),
                                                COMP(0))). TRY(T1).
     ...          R          // ADOP     ///  SCAN(1). TRY(AE0).
     ...        $( R         // $)       ///  PHRASE(COMP(0)). CYCLE.
                                              SCAN(1). TRY(F1).
     ... IDENT $= R          //                PHRASE(EMIT(STORE,COMP(2),
                                                COMP(0))). TRY(U1).
     OTHERWISE                                ERROR.

 T01 ... $TO IDENT           //                PHRASE(EMIT(GOTO,COMP(0))).
                                              TRY(U1).
     OTHERWISE                                ERROR.

STOP1 ...       $STOP        //                PHRASE(EMIT(STOPPER)). TRY(U1).
      OTHERWISE                               ERROR.
```

2-66

This schema will produce the same results as the first schema but makes no use of syntactic type. Herein lies the power of the productions analysis method: one can parse into phrases (schema III), a complete syntax tree (schema II) or mix the two (schema I).

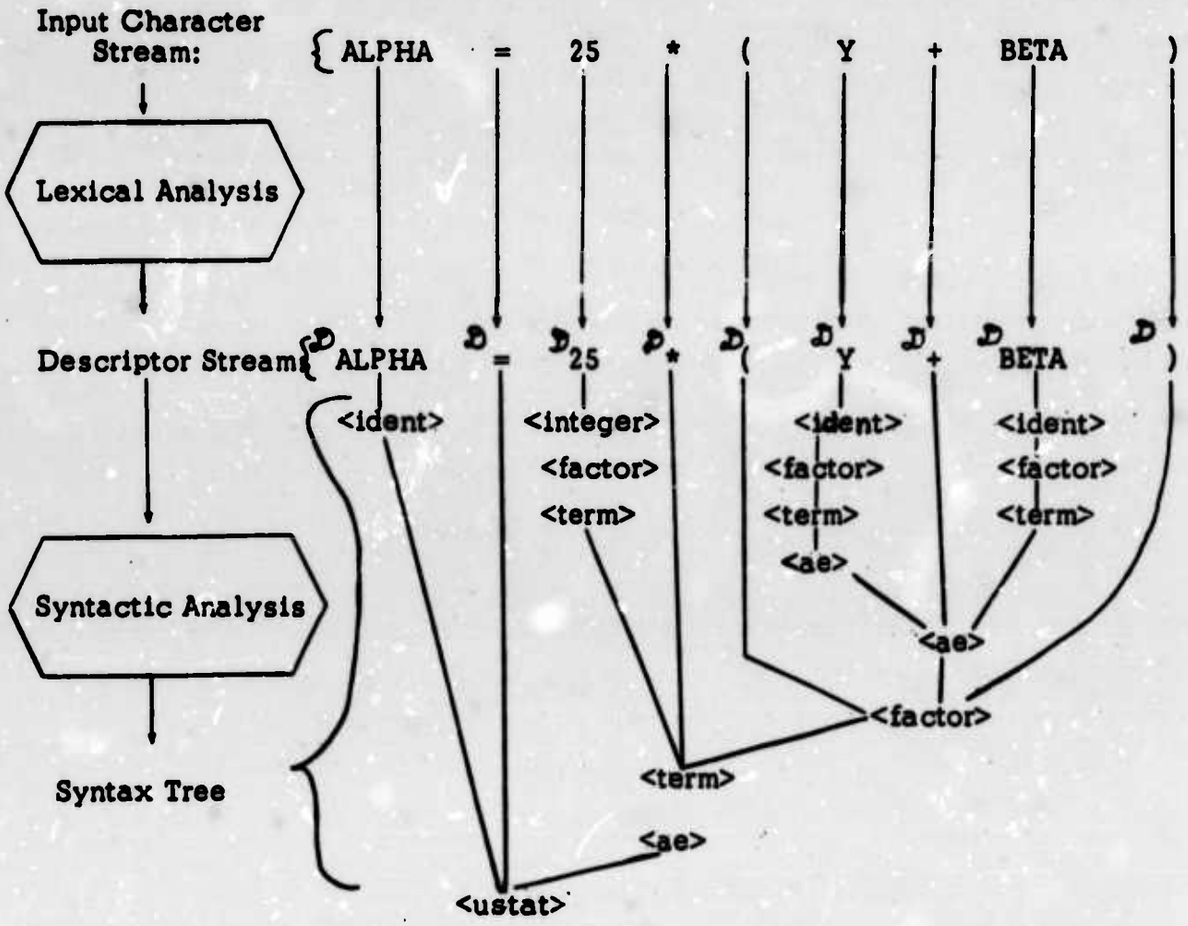
Now let us turn our attention to the problem of generating pseudo-code from the result of a predictive analysis. The Cheatham, Sattley paper in the appendices discusses various ways of tying the syntactic analysis and code generation together; at this point we will only discuss one technique of generating pseudo-code and assume that we are <u>given</u> a complete syntax tree.

Let us reconsider the example of section 2.0.

2-66

2-67

Analysis of Source into a Syntax Tree



2-68

Suppose that we have the following mechanism:

1. A control element which is at any point in time "pointing to" some node of the tree;
2. A means for naming nodes relative to the "current" node.
3. A "place" at each node to park the name of the "result" associated with that node.

4. A means for moving the control element to another node, and remembering the node to which we were pointing (i.e., a stack).

5. A means for moving the control element to the last node from which we transferred control (i.e., pop it off the stack).

Given the above tree with "current" node initially the <ustat>, we can sketch a "tree walk" and generation of pseudo-code as follows:

	Current Node	Action
1	<ustat>	Consider the third "son", the <ae>
2	<ae>	It is only a <term>, consider the <term>
3	<term>	Consider the left son, the <term>
4	<term>	Consider the only son, the <factor>
5	<factor>	Consider the only son, the <integer>
6	<integer>	R<integer>← $\mathfrak{D}_{25}$ (Read: "The 'result' of the 'integer' node is $\mathfrak{D}_{25}$)"; return to previous node.
7	<factor>	R<factor>←R<integer>; return to previous node.
8	<term>	R<term>←R<factor>; return to previous node.
9	<term>	Consider the right son, the <factor>
10	<factor>	The <factor> is of the form (<ae>); consider the second son, the <ae>.
11	<ae>	Consider the first son, the <ae>
12	<ae>	Consider the only son, the <term>

Etc., resulting in R<ae> ←...← $\mathfrak{D}_Y$ and return to previous node.

	Current Node	Action
13	<ae>	Consider the third son, the <term> Etc., resulting in R<term>←...←$\mathcal{D}_{BETA}$ and return to previous node.
14	<ae>	Both first and third son "considered" (i.e., 'results' are associated with them). Second son is +; therefore R<ae> ←EMIT (PLUS, son-1, son-3) [That is, R<ae>←EMIT (PLUS, Y, BETA)] Return to previous node.
15	<factor>	R<factor>←R<ae>; return to previous node.
16	<term>	R<term>←EMIT (TIMES, son-1, son-3); return to previous node.
17	<ae>	R<ae>←R<term>; return to previous node.
18	<ustat>	Consider the first son, the <ident>
19	<ident>	R<ident>←$\mathcal{D}_{ALPHA}$; return to previous node.
20	<ustat>	R<ustat>←EMIT (STORE, son-1, son-3)

2-70

Note that we have introduced the idea of "tree name" (e.g., son-1, etc.) as denoting both a place to go (turn the control element pointer to) and as denoting a value associated with a node (EMIT (PLUS, son-1, son-3)).

Now let us introduce the following:

> son-1, son-2, ... , etc. name tree nodes relative to the current node (in a fashion clear from context)

CONSIDER(N), where N is any tree name, causes the control element to turn to the node N and the current node to be placed on the stack.

EMIT( ) is as before except that it places the descriptor of the result in the "result field" of the current node.

COPY(N) copies the result of node N into the result field of the current node.

We now have a series of "pattern" and "actions" for walking the tree and generating code for the example above. The rules are:

Take the first structure applicable for the type node control is currently "on" and carry out the actions; when out of actions, return control to the previous node (i.e., top of the stack).

Thus:

Control On	and	Structure Is	then do the	Actions
<ustat>				CONSIDER(son-3) CONSIDER(son-1) EMIT(STORE,son-1,son-3)
<ae>		<term>		CONSIDER(son-1) COPY(son-1)
<ae>		<ae> + <term>		CONSIDER(son-1) CONSIDER(son-3) EMIT(PLUS,son-1,son-3)
<ae>		<ae> - <term>		CONSIDER(son-1) CONSIDER(son-3) EMIT(MINUS,son-1,son-3)
<term>		<factor>		CONSIDER(son-1) COPY(son-1)
<term>		<term> * <factor>		CONSIDER(son-1) CONSIDER(son-3) EMIT(TIMES,son-1,son-3)
<factor>		(<ae>)		CONSIDER(son-2) COPY(son-2)
<factor>		anything else		CONSIDER(son-1 of son-1) COPY(son-1 of son-1)

2-71

Note the more complicated tree name son-1 of son-1; the generalization to complex tree names is obvious. Thus, to get from the node $\mathcal{D}_{ALPHA}$ to the node $\mathcal{D}_{BETA}$ in the above syntax tree we could proceed to: son-1 of son-1 of son-1 of son-3 of son-2 of son-3 of son-1 of son-3 of father of father. Or turning the specification "around" and going by a different route: to father to right sibling to right sibling to son-1 to son-3 to son-2 to son-2 to right sibling to son-1 to son-1 to son-1. The Warshall and Shapiro paper in the appendices discusses a general purpose compiling system in which a rather elaborate tree walk and pseudo-code generation mechanism is available.

Another remark: Note the similarity between the patterns and associated actions for handling generation of pseudo-code from precedence analysis into phrases and the patterns and actions for processing the complete syntax tree. In both cases we have our "attention" somewhere (top of stack or 'current' node) and specify the context via a pattern which leads to certain actions. The similarity is more evident if we rewrite the "patterns" for the syntax tree as:

<ustat>
<ae> [<term>]
<ae> [<ae> + <term>]
etc.

denoting the tree structure by appropriate bracketing.

2-72

3.2 Handling Declarations

Fundamentally, declarations in a programming language are devices for dynamically (i.e., at compile time) changing the syntax of the source language. That is,

real X, Y;

really means something like: "add the two rules

<realvar>::= X
<realvar>::= Y

to the syntax".

In a practical compiler, however (or even, I would guess, in an impractical one), such declarations are generally handled by setting certain flags in the symbol (literal, etc.) table, so that the appropriate type information can be readily available for the processing of the parsed input into pseudo- (and machine-) code.

2-73

2-72

The handling of "shape" declarations (e.g., array A (I:20, J:40), overlay (W, V), etc.) depends to a considerable extent on the environment into which the result of compilation is going. Thus, if a full blown assembly is to take place on symbolic output from a compiler* then such declarations can usually be handled by invoking the "block started by symbol", "synonymous with", and the like pseudo operations usually available in an assembler.

Further, the handling of declarations which need to contact the environment, for example the declaration of library subroutine calls, filed data description references, and the like are highly dependent upon the environment in which the compiler is running in addition to the environment in which the compiled result will be placed.

---

* Rather a foolish way to do business, incidentally, if cost is of any importance in the long run.

We will not, at this time, go any further into the question of how we actually process the declarations except to make the following remarks:

1. By hook or by crook, the effect of the declaration must be reflected in the symbol (literal, etc.) table entry for the item declared.

2. Somehow, we must be able to snatch control from the syntactic analysis - code generation mechanism to handle the interpretation and storing of declarations.

3. While there exist reasonably elegant schemes for "automatically" doing syntactic analysis (and even much of code synthesis), the handling of declarations is generally messy with any but the simplest of languages. For this reason (among others) it will prove highly useful in any general purpose compiling system to have the ability to do arithmetic and relationals -- i.e., the "action language" should contain at least the rudiments of a good algebraic language.

2-74

For processing languages where several types of data can be manipulated (e.g., integer, real, boolean, string, etc.) it may be convenient to utilize the field, "TYPE", for carrying declarative information. Thus in this setting we will think of descriptors as quadruples:

(TABLE, LINE, LARG, TYPE)

where TYPE is any convenient encoding of which of the possible types the (described) value is.

3.3 Miscellaneous Manipulations

There are several different kinds of manipulations of source material which are more-or-less naturally done with the material in the form of a sequence of pseudo-instructions. Some of these (type conversion, scaling, and the like) are needed for languages which are "loose" in the sense that many detailed decisions in the compilation are left to the compiler; others (compile time computation, commutation of operands, and the like) are more to do with the generation of optimal coding; still others ("macro" expansion, and the like) are to do with both. We will discuss some of these and some mechanisms for handling them in the sections below.

2-75

3.3.1 Type Conversion; Scaling; Compile Time Computation

In languages which allow a variety of data types and which allow "mixing" these types in expressions (or don't allow it, but the compiler is trying to make a reasonable error recovery) we are faced with the problem of finding operations with operands of mixed types and effecting the conversion of one of the operands to the "preferred" type. This same problem, in a somewhat more complex form, arises in languages in which computations with scaled fixed point data are allowed (and it is assumed that the compiler tries to adjust scales appropriately for intermediate results) or in which data may have units (e.g., feet, miles/hr, etc.) and the compiler is expected to handle unit conversions of intermediate results.

In its simplest form the problem is simply this: for any simple computation (i.e., add, subtract, multiply, divide, store) in which the two arguments are of different types, the one of "less preferred" type is to be converted to the "more preferred" type prior to the computation being performed. Let us suppose for simplicity that a language allows integers and floating numbers and that integers are to be "floated" before an operation with floating numbers except that the "left hand side" type takes preference in assignment statements. Let us further suppose that there are two unary operators FLOAT, UNFLOAT which are available as pseudo-operations. Then the handling of this kind of conversion

fits into our "descriptor list" framework reasonably well. The example following suggests the mechanism required.

Suppose that we have a stream of descriptors representing the computation for an assignment statement and suppose that we can turn our attention to the "STORE" operation. The processing then to be performed is described by the table below. Here we have "patterns" and associated "actions". The non-obvious action is PROCESS ( ) which takes as argument the name of a descriptor and means: If the descriptor is not of a line in the descriptor table (i.e., "the argument is not 'complex' - it is a simple identifier or literal), then do nothing; otherwise remember which operation we are working on and turn our attention to the one designated, applying those actions appropriate; when finished copy the TYPE field of the line processed into the TYPE of the argument*. We further assume that the TYPE field of a descriptor is set appropriately for all variables and literals upon entry.

Processing for Simple Type Conversions

	Pattern			
	Operation	Arg-1**	Arg-2	

I	STORE	X	Y	PROCESS (Y);
				If TYPE (X) ≠ TYPE (Y) then
				if TYPE (X) = integer then
				Y←EMIT (FLOAT, Y)
				else Y←EMIT (UNFLOAT, Y);

II	ARITH	X	Y	PROCESS (X); PROCESS (Y);
				If TYPE (X) ≠ TYPE (Y) then
				if TYPE (X) = integer then
				X←EMIT (FLOAT, X)
				else Y←EMIT (FLOAT, Y);
				TYPE (ARITH)← float;

* Note the similarity with the CONSIDER ( ) action we used in section 3.1. .

** Read "Local name of first argument"

Suppose our old friend

$$\text{ALPHA} = 25 * (Y + \text{BETA}) .$$

were to be processed and that ALPHA, Y are floating and BETA (and, of course, 25) is an integer.

Below we have pictured the descriptor table before and after the processing (the column "step" indicates which step in the processing causes the change) and sketched a resumé of the processing carried out. (Recall the function EMIT ( ) from section 3.1; we assume for this example that line 105 of the descriptor table is initially the next available line.)

2-77

Descriptor Table

Line	Initial Contents	Type	Final Contents	Type	(Step)
⋮	⋮	⋮	⋮	⋮	⋮
66	PLUS	—	PLUS	floating	(3)
67	Y	floating	Y	floating	
68	*BETA	integer	*105		(3)
69	TIMES	—	TIMES	floating	(4)
70	25	integer	107		(4)
71	*66	—	*66	floating	(3)
72	STORE	—	STORE		
73	ALPHA	floating	ALPHA	floating	
74	*69	—	*69	floating	(4)
⋮					
105	——		FLOAT	floating	(3)
106	——		*BETA	integer	(3)
107	——		FLOAT	floating	(4)
108	——		*25	integer	(4)

Resumé of Processing

Step	Current Line	Current Operation	
1	72	$\mathcal{D}_{\text{STORE}}\ \mathcal{D}_{\text{ALPHA}}\ \mathcal{D}_{(69)}$ floating	Pattern I applies; The second argument, Y ~ $\mathcal{D}_{(69)}$, is complex, hence process it.
2	69	$\mathcal{D}_{\text{TIMES}}\ \mathcal{D}_{25}\ \mathcal{D}_{(66)}$ integer	Pattern II applies; The first argument is simple, hence do nothing, the second argument is complex, hence process it.
3	66	$\mathcal{D}_{\text{PLUS}}\ \mathcal{D}_{\text{Y}}\ \mathcal{D}_{\text{BETA}}$ floating integer	Pattern II applies; neither argument requires processing; the TYPE fields do not match and TYPE (X~$\mathcal{D}_{Y}$) ≠ integer, hence Y←EMIT (FLOAT, Y); TYPE (ARITH)← floating; return to considering line 69 and copy TYPE of floating into its' second argument.
4	69	$\mathcal{D}_{\text{TIMES}}\ \mathcal{D}_{25}\ \mathcal{D}_{(66)}$ integer floating	Types don't match; do X← EMIT (FLOAT, X); TYPE (ARITH)← floating; return to considering line 72 and copy TYPE of floating into its' second argument.
5	72	$\mathcal{D}_{\text{STORE}}\ \mathcal{D}_{\text{ALPHA}}\ \mathcal{D}_{(69)}$ floating floating	Types match; finished with processing.

2-78

The important point to be noted here is that the mechanism of "match the applicable pattern and perform the actions associated with it" seems a reasonable mechanism. The only difference between this and the mechanism for generating pseudo-code (section 1.3.1) is that the patterns are applied to operations rather than phrases; however, each is, you will note, merely a sequence of lines in the descriptor table.

A slight extension of this mechanism will handle the elimination of computation with constants by performing such computation at compile time*.

The more general type conversion problem cannot be handled by such a "local" mechanism. The scope of the problem is suggested by the following.

Suppose that we have the computation A = B* (C+D) + E* (F /G) with scaled fixed point numbers as follows:

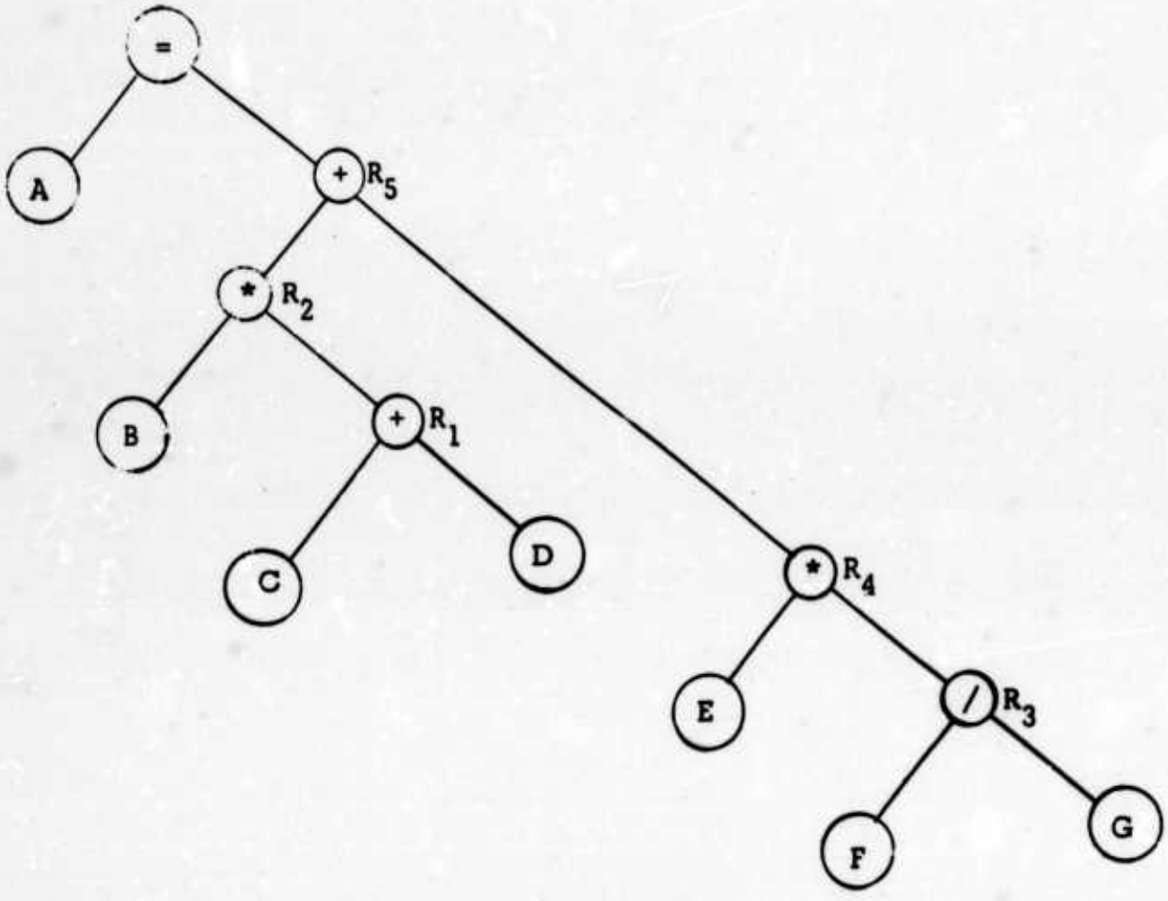
Variable	Scale	
	Bits to	
	Left	Right
A	16	4
B	8	1
C	14	8
D	3	2
E	16	6
F	8	5
G	5	0

2-79

---

* This is not an "unproblem". For example a language which allows.... PI IS 3.14159265 ... ZILCH = 2*PI+X .... as well as allowing general mapping functions which might contain constants, e.g., A(I) is accessed at word 2*I+512 of area based at β and a call for A(6) occurs. Generally, as languages allow more sophisticated declarations (particularly of computational fragments such as accessing (mapping) functions) the occurrences of a variety of redundancies of this kind will increase.

We might depict this computation as follows:



2-80

Here, $R_1, \ldots, R_5$ name the five intermediate results. The problem is to determine scaling for $R_1, \ldots, R_5$ in such a way that no overflow occurs* and that there are a minimum number of shift instructions required. The mechanism employed above (i.e., proceeding to the "innermost" computations and deciding on a scaling on the basis of local information only) could be used, but the number of shifts generated might be excessive. A better scheme is to provide a mechanism which can "wander around" this tree collecting known scaling information, postulating scales for the R_i, looking at the effect of postulated scale to change the scaling information, re-postulating scales, etc.

---

* or, a certain precision or significance is maintained, etc. depending upon the scaling philosophy employed.

We will only remark at this point that for this kind of problem we postulate an extension of the descriptors for carrying pertinent scaling information. That is, we now think of a descriptor as being <u>five</u> fields.

(TABLE, LINE, LARG, TYPE, LINK)

where LINK is a pointer to an "extension" of the descriptor which housekeeps (for example) scaling information. We do not consider adding such fields to the descriptor (permanently) as the need for such extensions is purely local. Thus we think of some small set of available extensions and link these to descriptors being processed and unlink them (back to being "available") when we are through with them*.

2-81

---

* We might note that the problem of generating highly efficient machine code is highly similar in that an extension to the descriptor is required (again locally in the same sense) to housekeep such information as which registers contain the item, position within register, sign description, etc.

3.3.2 "Macro" Expansion

While the idea of "macro assemblers" has considerable stature (one is included in any respectable software package these days) the idea of allowing similar macro facilities in a compiler is found in few, if any, systems. However, the basic need in most any compiler is clear since, for example, accessing an array really requires at least an implicit macro. That is, the fragment

... A(I, J + 2) * K ...

where, for example, A(,) has been declared as a 10 X 10 array is, really, in terms of pseudo-operations, something like:

```
                    ⋮
   α        D PLUS D J D*2
  α + 3     D TIMES D I D*10
  α + 6     D PLUS D@ D*(α+3)
  α + 9     D LOCATE D A D*(α+6)
  α +12     D TIMES D(α+9) D*K
                    ⋮
```

That is, we must compile coding to compute the quantity (10*I) + J + 2 and add this to the base address (to within a constant) to "locate" the word containing the item in order to access A(I, J+2).

In most compilers, such data accessing is handled by one or more special purpose bumps on the compiler.

It is a quite straight forward addition to a compiler to allow for "macros" of pseudo-operations to handle data accessing plus other kinds of operations and, incidentally, to allow the programmer to define, in source language, macros and to call them later in his program (or, if he is operating in a reasonable programming system with a reasonable filing sub-system, to call them later in

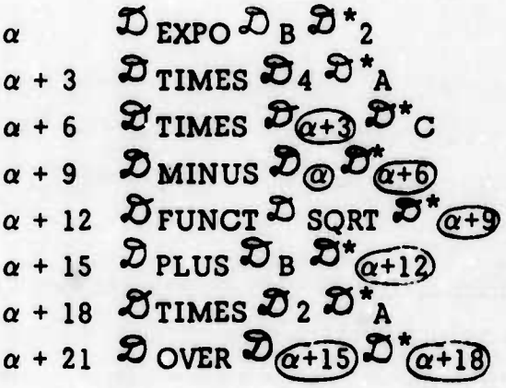
2-82

some other program). Thus, one could write

define macro ROOT1(A, B, C) AS (B+SQRT(B**2-4*A*C)/ (2*A);

and use this ROOT1 (,,) later in his program, assured that "open" coding appropriately optimized would result.

The mechanism for handling this is really quite straight-forward. The compiler produces the (pseudo-operations) descriptor list fragment for the macro, for example:



2-83

and then "parks" this list somewhere effecting a few changes, to wit: replace $\mathcal{D}_A, \mathcal{D}_B, \mathcal{D}_C$ with $\mathcal{D}_{[1]}, \mathcal{D}_{[2]}, \mathcal{D}_{[3]}$ and replace $\mathcal{D}_{(1)}$ with some appropriate "local" or "self" relative descriptor. Thus the symbol table entry for ROOT1 is tagged to indicate that it is a macro (with three arguments). A call for, for example, ROOT1 (4,X+Y*Z,Z) later in the program would result in the above (modified) descriptor list to be brought out and the $\mathcal{D}_{(1)}$ and local relative reference descriptions to be appropriately modified. Thus, the call might result in:

δ	TIMES Y Z
δ + 3	PLUS X δ
δ + 6	EXPO δ+3 2
δ + 9	TIMES 4 4
δ + 12	TIMES δ+9 Z
δ + 15	MINUS δ+6 δ +12
δ + 18	FUNCT SQRT δ+15
δ + 21	PLUS δ+3 δ +18
δ + 24	TIMES 2 4
δ + 27	OVER δ+21 δ +24

Note that the later optimization, for example detecting and removing constant computations would then eliminate lines (starting with) δ+9, δ+24 and could also result in δ+27 being replaced by TIMES δ+21 1/8 saving some time (on most computers).

A few more remarks about data and data accessing are pertinent. If the accessing of data is viewed as a sequence of computations culminating in a "locate" operation for locating the word in memory (usually relative to some base address) and if this computation sequence is viewed as a macro in the above sense, then the actual computation required to perform this "locate" can be as complex as desired without this being any strain on the compiler at all.

2-84

One kind of application is the following: Suppose we had a very long vector, V(I); the mapping function could be:

```
LOCV..   J = I/10000;
     if  J = CURBLOCK  then
         RESULT = LOCATE (V, I - 10000*J)
     else begin  OUTPUT (V, 10000, CURBLOCK);
                 INPUT  (V, 10000, J);
                 CURBLOCK = J; to (LOCV) end
```

Here we are presuming INPUT/OUTPUT (i, j, k) causes input/output of the

material based at location i for j locations to be input from/output to block k of some tape, disc, magnetic card, etc. file.

One more idea on data accessing. In order to handle the accessing of data items packed several per word we can postulate a pseudo-operation FIELD taking three arguments: "address" of a locate, and two (integer) values. The interpretation of FIELD(L, I, J) is: consider the word located and take (fetch or store, depending on context) the field starting at bit I and being J bits long. We will not pursue this further here.

2-85

3.4 Handling Special Features

Most programming languages have one or more rather annoying features, annoying in the sense that the handling of them often requires some kind of special bump on the compiler. A very brief sketch of some of these follows:

3.4.1. Variable Remote Connections

Many languages allow one to set certain variables to "addresses" rather than values. That is, one (usually not in this format) can write:

```
    V:=L;
    to (V);
L.. X:=Y+Z; to (C);
M..X:=Y-Z;
C..
```

2-86

The annoyance here is that the collection of labels which can be so referenced must each have an associated address (to allow the V:=L to be set). There are two basically different ways to implement the above, below sketched in SAP or FAP like 7094 coding:

METHOD I	METHOD II
⋮	⋮
CLA L	CLA .L
STA V	STA V
⋮	⋮
TRA V	TRA V
L NOP L	L CLA Y
CLA Y	ADD Z
ADD Z	STO X
STO X	TRA C
TRA C	M CLA Y
M NOP M	SUB Z
CLA Y	STO X
SUB Z	C etc
STO X	⋮
C etc	V TRA error
⋮	.L TRA L
V TRA error	.M TRA M

2-87

(Note: We realize that one can get a bit cute with the use of indirect addressing, but the examples are to make another point.)

Method I is simpler from the compiler's point of view but has an extra useless instruction following every label (which can be used as the argument of such a connection); Method II requires that L be interpreted in two different ways (as L or .L) depending on its' usage.

It should be remarked that if the coding is to be placed in any sophisticated programming system or if clever optimization (using flow analysis) is to be performed, the whole idea of variable remote connections is a bad one and should be replaced by a switch mechanism.

3.4.2 Status Constants

JOVIAL introduced the idea of status constants. A status constant is an identifier declared to have a literal (indeed, integer) value which depends upon which status variable of which it is a value. That is, one can declare (in effect):

```
STATE  status  (OHIO, NY, PA, MASS);
CITY   status  (LIMA, BOS, PHILA, NY)
```

Very roughly, in JOVIAL the identifiers OHIO, NY, PA, MASS are treated as, respectively, literal 0, 1, 2, 3 when they appear in an appropriate context (e.g., STATE:=MASS, if STATE = MASS, etc.); similarly, LIMA, BOS, PHILA, NY are literal 0, 1, 2, 3 in appropriate context.

The problem here is interpreting the appropriate context in the case of NY which is 1 or 3 depending on whether STATE or CITY is hard by. This interpretation can be done in a variety of ways; the use of a "syntax tree walk" is quite reasonable as is an inspection of the pseudo-code sequence.

2-98

3.4.3 Complex Data Structure

COBOL was the first major language to introduce the idea of highly structured data (in the sense of field overlay, or, put better, in the sense of handling n-tuples). The "new programming language" (now called PL/I) being devised for System/360 has somewhat more along this line, allowing for data elements which are defined as n-tuples of previously defined data elements or of arrays of previously defined data elements, and so on, all recursively. The basic problem in handling such constructs is that a given name may stand for a highly structured collection of (atomic or basic or directly manipulable, etc.) data elements. To handle such things one must, by one method or another, attach to each "entity" a structural description (kind of a generalization of a type code) which is manipulable. Such a description would probably take the form of a tree.

4 GENERATION OF MACHINE CODE

4.0 General Discussion

Let us recapitulate our assumptions about "what has happened" and "where we are" when we consider the generation of the output coding. The source program has been parsed and the parse "interpreted" into a series of pseudo-instructions. These pseudo-instructions have then been amended and extended until they represent the sequence of computations to be performed and are consistent with respect to operand types and the like.

The problem of generating (good) machine code can be thought of as a three stage process. First, the pseudo-code is analyzed in order to determine domains of invariance for each of the variables in the program. Given this, the pseudo-code can be put in some canonical form (e.g., commute all commutative operands to a standard order, and the like), sub-computations which are identical can be eliminated, and sub-computations which are invariant in a "loop" can be removed to the outside of the loop. Then, the resulting pseudo-code can be analyzed and the various members of classes of special registers can be allocated to certain variables or sub-computations over various "regions" of the code. Finally, the machine code can be generated and "handed on" to whatever processor is to dispose of it (e.g., formation of relocatable binary (or decimal, etc.), absolute binary, symbolic for future assembly, and so on).

2-89

2-98

Few, if any, current compilers have all these facilities for the generation of coding; however, if highly optimal code is desired, such facilities must be available. Of course if a compiler is properly built, it should be possible to bypass any of these "optimization" sections if a "quick and dirty" compilation (for debug purposes, for example) is desired.

4.1 Analysis of Pseudo-Code

The first step in generating highly efficient machine code is to perform an analysis of the flow structure and, in particular, the domains over which the

---

* Again we emphasize that this much has not necessarily been done -- e.g., the type consistency may not be guaranteed; one must tailor any compiler to the particular language-machine-environment triple.

various quantities manipulated by the program are invariant. Until this information is available it is impossible to do very much toward eliminating common computations, removing invariant computations from loops, allocating special registers, and so on.

The only completely satisfactory way to carry out this analysis is to develop, for each variable (and hence each sub-computation) a complete picture of those areas of the program over which the variable may possibly have different values. A complete knowledge of the flow structure of the program is required for this, however. If the complete analysis of the flow structure is considered too expensive then a much simpler technique is to consider the "flow blocks" in a program; that is, those computation sequences into which control flows only at the "top" and which can transfer control only at the "bottom". If one makes no assumptions about a variable outside a flow block and simply divides the flow block into its domains of invariance for each variable, then common sub-computations, invariant computations, and the like can still be handled only on this more "local" basis. This latter technique is extremely "cheap" in terms of both the amount of code required to collect the information as well as the compile time required to carry it out. The Warshall-Shapiro paper in the appendix alludes to a specific technique of this type actually implemented in a compiler and the bibliography there will lead on to the details of such a scheme.

It should be remarked here that if one is doing a thorough analysis then the notion of "do loop" or "for loop" should be handled in the same manner as if the programmer had written out the loop control instructions. That is, it should be the data resulting from a complete analysis which controls the optimization decisions, not the fact that a certain area has been designated a loop with loop variable "I" by the programmer. Indeed it may turn out that the item which is really "controlling" the loop is 10*I or I + 6 or Z*I + 5. etc., this information will be developed from a complete analysis.

4.2 Invariant Computations and Common Sub-Expressions

Given the analysis resulting in the invariance domains for the variables of the program we can then re-order the program to eliminate computations appearing in a "loop" context which do not depend on the loop variables and eliminate computations which are common, performing them only once. First, however, if certain pseudo-instructions have commutative operands (and the source language or local conventions allow re-ordering the computations within an expression) they should be "flipped" into some canonical order so that comparisons for common computations and the like can be made more easily.

Consider the program:

```
            I:=2 ; A(1):= 1;
HENRY:      A(I):= I;
      for   J:= 1 step 1 until 10
        begin   B(J): = A(I+1);
                C(J):= D(J) + A(I+1) end
            I:=I+1;
      if    I<50 to HENRY else STOP
```

Let us suppose that the representation in pseudo-code is the following:

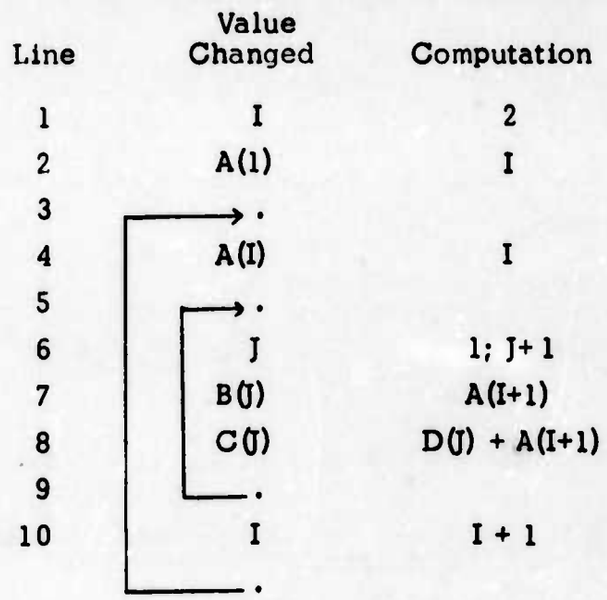
2-90

2-91

```
Line    Operation       Arguments
  1     STORE           I       2
  4     LOCATE          BASE(A) 1
  7     STORE           (4)     1
 10     LABEL           HENRY
 12     LOCATE          BASE(A) I
 15     STORE           (12)    I
 18     STORE           J       1
 21     LABEL           .1
 23     LOCATE          BASE(B) J
 26     FIELD           (23)    FIRST(B)  NO(B)
 30     PLUS            I       1
 33     LOCATE          BASE(A) (30)
 36     STORE           (26)    (33)
 39     LOCATE          BASE(C) J
 42     FIELD           (39)    FIRST(C)  NO(C)
 46     LOCATE          BASE(D) J
 49     FIELD           (46)    FIRST(D)  NO(D)
 53     PLUS            I       1
 56     LOCATE          BASE(A) (53)
 59     PLUS            (49)    (56)
 62     STORE           (42)    (59)
 65     PLUS            J       1
 68     STORE           J       (65)
 71     LESS            J       10  .1  .2
 76     LABEL           .2
 78     PLUS            I       1
 81     STORE           I       (78)
 84     LESS            I       50  HENRY  .3
 89     LABEL           .3
 91     STOP
```

We are assuming here that the variables B, C, D are packed into some (bit) field of a word.

We might depict the result of analyzing the invariance domains as follows:



Line	Value Changed	Computation
1	I	2
2	A(1)	I
3	→ .	
4	A(I)	I
5	→ .	
6	J	1; J+1
7	B(J)	A(I+1)
8	C(J)	D(J) + A(I+1)
9	.	
10	I	I + 1
	.	

We note, for example, that the area enclosed by steps 5-9 does not epend upon I or A( ), etc.

Let us suppose that:

$$\text{BASE}(A) = \alpha$$
$$\text{BASE}(B) = \text{BASE}(C) = \text{BASE}(D) = \beta$$
$$\text{FIRST}(B) = 1 \quad \text{NO}(B) = 6$$
$$\text{FIRST}(C) = 7 \quad \text{NO}(C) = 12$$
$$\text{FIRST}(D) = 19 \quad \text{NO}(D) = 12$$

Then the common computations are lines

23, 39, 46
30, 53, 78
33, 56

The invariant computations are:

Lines	Move to
30, 33	Between 18 and 21
53, 56	
78, 81	

The re-ordering of computations and elimination of common computations could result in the following:

2-94

Line	Thread	Common	Operation	Arguments			
—	1		—				
1	7		STORE	I	2		
4			LOCATE	α	1		
7	10		STORE	(4)	1		
10	15		LABEL	HENRY			
12			LOCATE	α	I		
15	18		STORE	(12)	I		
18	81		STORE	J	1		
21	36		LABEL	.1			
23		*	LOCATE	β	J		
26			FIELD	(23)	1	6	
30		*	PLUS	I	1		
33		*	LOCATE	α	(30)		
36	62		STORE	(26)	(33)		
39			—				
42			FIELD	(23)	7	12	
46			—				
49			FIELD	(23)	19	12	
53			—				
56			—				
59			PLUS	(49)	(33)		
62	68		STORE	(42)	(59)		
65			PLUS	J	1		
68	71		STORE	J	(65)		
71	76		LESS	J	10	.1	.2
76	84		LABEL	.2			
78			—				
81	21		STORE	I	(30)		
84	89		LESS	I	50	HENRY	.3
89	91		LABEL	.3			
91			STOP				

The "thread" gives the new sequencing of the computation; computations which are common are marked with a *.

The interpretation is given by the following:

Step	Line	Computation
1	1	STORE I 2
2	7	STORE [LOCATE α 1] 1
3	10	LABEL HENRY
4	15	STORE [LOCATE α I] I
5	18	STORE J 1
6	81	$C_1 \equiv$ PLUS I 1
		STORE I C_1
7	21	LABEL .1
8	36	$C_2 \equiv$ LOCATE β j
		$C_3 \equiv$ LOCATE α C_1
		STORE [FIELD C_2 1 6] C_3
9	62	STORE [FIELD C_2 7 12] [PLUS [FIELD C_2 19 12] C_3]
10	68	STORE J [PLUS J L]
11	71	LESS J 10 .1 .2
12	76	LABEL .2
13	84	LESS I 50 HENRY .3
14	89	LABEL .3
15	91	STOP

2-96

The computations $C_i \equiv$ ... are read "compute the quantity and park it somewhere as the $i^{\underline{th}}$ common computation to be used later".

It should be noted that the reordering of the computation (in terms of common sub-expression computations) makes the generation of coding for a machine with a stack more difficult in that to "park" common computations in the stack brings up a rather messy stack accounting problem during code selection (especially if the stack is "finite").

4.3 Special Register Allocation

Given the threaded computation sequence as above there remains one problem before the generation of final machine code can be performed. If the computer for which code is to be generated has index registers, base address registers, a collection of accumulators, limit · gisters, multiple instruction registers, or the like it is necessary to determine what quantities are to reside in each of these registers at what times.

One scheme for performing this "reservation" of registers for various quantities is roughly the following: We associate with each argument of each pseudo-instruction information indicating the "affinity" of that argument for certain registers*. For example, the first argument of a LOCATE has an affinity for a "base address" register and the second argument has an affinity for an "index" register.

Note that we are not assuming that variables are necessarily the objects allocated to registers; rather any computation which can be usefully kept in a special register should be considered (especially common computations).

Given these "affinities" and the computation sequence it is reasonably straight forward to apply an algorithm which computes the "cost" of keeping and not keeping these quantities in special registers over some convenient chunk** of the program. Given these cost figures an allocation can then be made resulting in a list of quantities to be loaded into and maintained in certain registers over certain regions of computation. (Presumably certain members of each class of special registers will not have allocated quantities in them but will be available for "junk" usage by the code selection machinery.)

---

* Note: It should be clear that we are leaving out the "one of a kind" arithmetic registers (accumulator, quotient, etc.); this allocation is more properly part of the detailed code selection.

** Determining what is a "convenient chunk" is a decidedly non-trivial problem which can be solved only by looking at the connectivity amongst the various flow blocks of a computation. Taking blocks (in the ALGOL sense), loops, and the like as such chunks often yields a reasonably good approximation.

4.4 Code Generation

Given the threaded computation sequence and the register allocation information we are finally in the position to generate coding.

It is convenient to think of an "extension" to the descriptors composing the pseudo-code to where we can park detailed information concerning the status of the argument or computation represented by the descriptor: what register (s) contain the item; whether the sign is correct; and so on. The LINK field was added to the descriptors for this purpose (see section 3.3.1). Further, it is convenient to have a "track" table with an entry for each special register in which is indicated what, if any, computation is currently residing in that register.

Basically, then, one simply turns his attention to a required computation (following the thread) and (recursively) to the sub-computations required for that computation, etc. The specifics of "how" to generate code are, of course, highly dependent upon the particular hardware for which code is being generated. We will thus not pursue further "how" to generate code but merely make pseudo-operation and its' second argument is tagged as (satisfies the predicate) as being currently in the accumulator.

In the event that the number of patterns required to discriminate the various interesting possibilities grows too large, it is convenient to allow "questions" (i.e., if statements) in the actions (as was implied in section 3.3.1). A few remarks relating this to the machinery we discussed above are appropriate:

1. The actual generation of code once it has been determined what code is to be generated can be handled by the EMIT ( ) mechanism introduced in section 3.3.

2. The pattern-action idea is applicable here also; indeed the problem is to, starting with a desired computation, to PROCESS ( ) or CONSIDER ( ) its' operands (recursively walking down the computation tree). At any point one is interested in discriminating the pseudo-code via patterns like

PLUS X accumulator
PLUS index constant

and so on. By adding to the possible elements of a pattern a "predicate" mechanism such discriminations are easily made. Thus, the pattern

PLUS X accumulator

is interpreted: Our attention is on a PLUS whose second argument is in the accumulator and whose first argument may be anywhere.

2-98

2-99

5. CONTACTING THE ENVIRONMENT

5.0. General Discussion

There are three senses in which we must discuss contacting the environment:

1. The compiler contacting its environment to procure library items (descriptions, macros, and the like).
2. The compiler attending to arranging that the output for coding will contact its environment (appropriate calling sequence structuring, indication of library subroutine usage, need for peripheral equipment, and so on).
3. The compiler "handing over" its' resultant output to the environment (as symbolic coding, relocatable binary, and so on).

We would like to distinguish three basically different kinds of environments. The most simple would be that of a "barefoot" computer.

The next might be a computer with a conventional (batch processing) monitor. Finally, we might have a computer with a full scale modern programming system.

Finally, we would like to discuss three types of linkage which must be effected by a compiler and/or environment. These are:

1. Linkage of programs to other programs (subroutines, called procedures, "system" routines and so on).
2. Linkage of programs to data.
3. Linkage of programs to hardware (tape drives, disc files, and so on).

We will not discuss these topics in much detail here. The Cheatham-Leonard paper and the Leonard-Goodroe paper in the appendix pretty well describe our attitude in these matters. In addition to these papers we will consider in detail only the problem of program to data linkage in these notes, this in section 5.1.

2-100

2-101

5.1 Program to Data Linkage

5.1.0 General Discussion

In most programming languages, the data which a program accesses is thought of as "part of" the program -- it is defined (structurally) as part of the program definition and is input or output as part of the program execution. The idea of "common" is introduced to allow two or more programs, compiled separately, to access the same information, or one program to reuse storage for new data it requires. Further, in most programming languages, the structures which data can assume are usually restricted to regular arrays and the smallest unit of data is the machine word.

This state of affairs is woefully inadequate for a large number of applications. For example:

1. In any program in which more than a handful of programmers are involved (for example command and control systems or management control systems or programming systems) the use of "common" areas for communication among programmers is completely inadequate unless there are more managers than there are programmers.

2. In most "system" programming it is necessary for reasons of space restrictions to be able to "pack" several items in one computer word; in many command and control systems data packed into words is "forced" into the system by devices such as radars and other sensors.

3. In many applications (information retrieval, "systems" programming) it is useful to be able to deal with data structures more complex than simple arrays (trees, lists, and the like, for example).

There have developed a couple of schools of thought on the handling of "global" data structures. The first of these is the COMPOOL facility first introduced (to the best of my knowledge) in the Lincoln Compiler in 1953 by

the Lincoln Laboratories. The various JOVIAL "systems" (as distinct from the JOVIAL language) also have a COMPOOL facility. The basic idea here is that all data structures (and, for that matter, programs) which are common to several programs are defined as part of a COMmunications POOL and a description of this COMPOOL is automatically made available to the compiler. The CL-I, CL-II, and BNX Programming Systems have facilities for filing data structure descriptions as well as data set instances and allow a declaration, in programs, to the effect that certain (named) data structures are to be referenced by the program. At one extreme (only one big data structure) this is equivalent to COMPOOL; however, generally a given program references only a few of the totality of data structures, so that one need declare only those actually referenced. Further, with these systems the collecting together of code and data for test or simulation runs is usually easier in that only these structures referenced need be available in memory.

2-103

2-102

5.1.1 Data Descriptions

5.1.1.0 General Discussion

By a data description we mean a body of information which includes one or more of the following:

1. Information allowing a compiler to generate coding to access an element of the data set.
2. Information allowing a (general purpose) data input/output package to input or output a set of values for the data set.
3. Information allowing a "debugging monitor" to (request a compiler to, perhaps) check values of data elements for validity (fall within some numerical range, have some particular relationship to some other data element(s), and so on).
4. Narrative text allowing a user (through some retrieval mechanism) to obtain a (English) description of the function, usage, and the like of the data set and/or specific elements of the data set.

2-104

2-105

We should remark that, from the point of view of a compiler, a data description is nothing more-or-less than a junior symbol table, literal table, and macro description table, all of which get "plugged" into the compiler when we are to reference any elements of the data set. In order to allow the kinds of manipulation outlines in points 2-4 above we require a bit more linkage plus a table of "narrative text".

In section 5.1.1.1 we discuss the elements of a data set and the body of information required concerning them in order to satisfy the above needs; in section 5.1.1.2 we consider the combination of elements into structures, these into larger structures, and so on.

5.1.1.1 Data Elements

A data element, in the sense in which we are currently employing the term, is an item which may be referenced (in an appropriate representation) via a statement in a programming language. A data element has:

1. A name, and a "form" of reference.

   For example, a vector component named V and referenced by "V (<ae>)"; or a vector component named W referenced by the form "W", and so on).

2. A type.

   For example, integer, floating, fixed with four bits right of the binary point, string, and so on).

3. A mapping function*.

   For example,

   FIELD(LOCATE (base (V), index*100+4), 1, 6)

   if A(index) = 0 then LOCATE (base (V), index)
   else LOCATE (base (V), L (index));

   and so on.

4. Constraints

   For example,

   0 <V(I) < I**2 + 1
   -50.6 ≤ V(I) ≤ 3.4567

   and so on.

---

* It may be that there are several forms of reference and, further, that the mapping function depends upon the form of reference and the context. For example, let S name a stack. Then S=5, A=S, S(1)=5, A=S(2) might all be allowable references with the first two involving push-down and pop-up and the last two considering the stack like a vector.

5. Units of representation
For example,

MILES PER HOUR
FEET PER SECOND PER SECOND
DYNES PER FORTNIGHT

and so on.

6. Miscellaneous
For example,

ROUND before truncating
SIGN to be carried on right (left, not at all)
ACCESSED only by people named SMITH
CONSTANT, with value 3.14159265. . . . .
PICTURE: SDDDPDD, etc.

2-106

2-107

5...1.2 Data Structures

1. Atomic elements
2. Collections by forming:
   - arrays
   - n-tuples
   - ordered sets
3. Apply (2) recursively
4. External and Internal Representations

   Card Formats - n-tuple plus indices plus identification appearing together on card.

THE TGS-II TRANSLATOR GENERATOR SYSTEM

T. E. Cheatham, Jr.
Computer Associates, Inc.
Wakefield, Massachusetts

2-111

CA-6505-2611
May 1965

To be presented at the International Federation for Information Processing (IFIP) Congress, May 1965.

THE TGS-II TRANSLATOR GENERATOR SYSTEM

T. E. Cheatham, Jr.
Computer Associates, Inc.
Wakefield, Massachusetts (USA)

Translator Generator System II, TGS-II, represents our "current position" in a project which has been underway for several years and which will probably continue for several more. The overall goals of this work have been to develop a general purpose compiling system which

a) is efficient as a compiler,
b) allows the generation of efficient machine code,
c) accommodates a variety of programming languages,
d) accommodates a variety of object computers (or other interpreters),
e) allows the rapid construction and documentation of a compiler for any specific language-machine pair, and
f) allows efficient implementation and documentation of modifications or extensions to languages and/or to the kind of code generated.

2-113

Over the past several years there have been a number of people who have contributed to the body of ideas and experience which have resulted in TGS-II; the bibliography lists several of the papers which have resulted [1, 3, 4, 5, 6, 9, 10, 11] from this work. We cannot conceivably even sketch the history or current status of this project in this paper. Rather, we will try to provide the flavor of the main body of our current ideas, leaving the details to future publications.

It is an accepted fact that the construction of a compiler for a reasonably simple programming language to produce straightforward code for a reasonable computer is not a technically difficult task at the present time. When, however, one adds sufficient facilities to a language (for example: scaled fixed point computations; data sets with more structural variability than number of dimensions, or whose elements may violate word boundaries;

context dependent interpretation of symbols; and so on) or when one is producing code which will be run in a sophisticated environment (having, for example: completely dynamic allocation and re-allocation of code, data, or other resources; COMPOOL or other global declaration facilities; real-time requirements; and the like) or when one demands highly efficient object code, the problem may become technically difficult rather quickly. There are today a host of "compiler generator" systems to which one describes the syntax of a language with, perhaps, fragments of coding and other semantics attached to various syntactic types, and which will then accept source strings in the language described and produce symbolic or binary machine coding [2, 4, 7, 8, 10, 11]. TGS-II is not such a system; rather, TGS-II is an environment in which the processing required of a particular compiler can be specified. It is our hope that the mechanisms available in TGS-II provide the compiler writer a framework which allows him to concentrate upon the strategy of compilation rather than the details of some particular host computer. We will try to suggest some of the mechanisms available in TGS-II in the paragraphs below.

TGS-II consists of the TRANDIR language and the TRANGEN system, essentially an interpreter which executes programs written in the TRANDIR language -- programs which describe a specific translation process.

The data which is processed by TGS-II consists of values and symbol descriptors. A symbol descriptor is a sextuple composed of three pairs of values: a pair, table code and line within the table in which the attributes of the symbol described are stored; a pair of control bits for marking interesting subsequences; and, a pair of fields whose specific function varies and might be forward or backward links, arithmetic type code, syntactic type code, number of users, allocated address and so on. The value processed by TGS-II may be literals; variables which are organized as scalars, arrays, or stacks; elements of tables; and fields of symbol descriptors. The number and format of the tables used in any TRANDIR program is largely up to the user; however, there are built into the system such tables as character

table, symbol table, literal table, terminal symbol table, operation code table, implicit label table, the table containing the sequences of symbol descriptors being processed, and a few others. Let us think of the translation process as taking place in five phases (not "passes" in the conventional sense), namely: lexical analysis, syntactic analysis or parsing, interpretation of the parse, sequencing and optimization of the computation, and code selection.

The material being processed in TGS-II is represented, at any point in the processing, as a sequence of symbol descriptors. Thus, the steps in a translation might be viewed as follows:

source sequence of character descriptors

↓

LEXICAL ANALYSIS

↓

sequence of identifier, literal, and terminal descriptors

↓

SYNTACTIC ANALYSIS

↓

sequence of phrase descriptors

↓

INTERPRETATION OF PARSE

↓

sequence of pseudo-code descriptors

↓

OPTIMIZATION ANALYSIS

↓

sequence of linked pseudo-code descriptors

↓

CODE SELECTION

↓

sequence of machine code descriptors

The manipulations performed during each of these phases is described in the one language, TRANDIR. Switching among these phases is thus completely straightforward, allowing one to produce a one-pass, two-pass, etc. compiler depending only on size considerations and amount of analysis for optimization desired.

The TRANDIR language may be thought of as a conventional algebraic language with imperatives for the normal arithmetic, relational and data moving operations for manipulating values and symbol descriptors to which have been added quite powerful declarative and "pattern testing" facilities. A pattern is a list of predicates each of which is applied to a symbol descriptor. The definition of the predicates, layout of various tables, and so on are included in the declarations which form a part of a TRANDIR program. Typical patterns would be

```
... ARITH.EXPR    ADD.OPERATOR    TERM / /
    ... IDENTIFIER    ".." / /
    / / PLUS    ACCUMULATOR    MEMORY ...
    / / MINUS    MEMORY    INDEXED.MEMORY ...
    / / TIMES    LITERAL    COMPUTATION ...
```

The double slash and ellipsis delimitors (plus others) indicate the particular sequence of several possible sequences currently being "pointed to" to which the pattern is to be applied. Thus, the TRANDIR programmer may utilize an exhaustive list of patterns to perform an analysis or may revert to the ALGOL-like if-then-else statements or he may choose to mix the two.

TGS-II is presently operational on the IBM 7094-II computer and will soon be available in the CDC-1604, UNIVAC M-490, and GE-635 computers. At the present time TGS-II has been used in the construction of translators for the TRANDIR language, a dialect of NPL, and a dialect of ALGOL. Presently, translators for Oak Ridge ALGOL, FORTRAN-IV, and a data description language are being developed.

In spite of the enthusiastic claims to the contrary, we feel that there is no such thing as an "instant compiler", unless the language and, particularly, the resulting code are simplified to the point of being uninteresting. However, we have found that with TGS-II the cost of construction and documentation of translators can be cut substantially (perhaps by a factor of two or more) and the cost of subsequent modification and extension is cut even further.

BIBLIOGRAPHY

1. R. Bolduc, T. E. Cheatham, Jr., A. L. Dean, Jr., "Preliminary Description of the Translator Generator System-I", CAD-64-3-SD, Computer Associates, Inc., April 1964.

2. R. A. Brooker and D. Morris, "An Assembly Program for a Phrase Structure Language", The Computer Journal, vol. 3 (1960), p. 168.

3. T. E. Cheatham, Jr., A. L. Dean, Jr., A. T. Dean, S. A. Schuman, "Preliminary Description of the Translator Generator System-II", CA-64-1-SD, Computer Associates, Inc., August 1964.

4. T. E. Cheatham, Jr. and Kirk Sattley, "Syntax-Directed Compiling", AFIPS Conference Proceedings Vol. 25, SJCC, Spring 1964, p. 31.

5. T. E. Cheatham, Jr. and S. Warshall, "Translation of Retrieval Requests Couched in a Semi-Formal English Like Language", Comm. ACM, Vol. 5, No. 1, January 1962, p. 34.

6. R. W. Floyd, "Syntactic Analysis and Operator Precedence", Jnl. ACM, Vol. 10 (1963), p. 316.

7. E. T. Irons, "A Syntax Directed Compiler for ALGOL-60", Comm. ACM 4 (1961), 51-55.

8. B. M. Leavenworth, "FORTRAN IV as a Syntax Language", Comm. ACM, Vol. 7, No. 2, February 1964, p. 72.

9. R. M. Shapiro and L. Zand, "A Description of the Input Language for the Compiler Generator System", CAD-63-1-SD, Computer Associates, Inc., June 1963.

10. S. Warshall, "A Syntax Directed Generater", Proc. EJCC, 1961, Macmillan & Co., 1961.

11. S. Warshall and R. M. Shapiro, "A General Purpose Table Driven Compiler", AFIPS Conference Proceedings Vol. 25, SJCC, Spring, 1964, p. 59.

AMBIT: A PROGRAMMING LANGUAGE FOR ALGEBRAIC SYMBOL MANIPULATION

by

Carlos Christensen

CA-64-4-R

October 15, 1964

The research reported in this paper was sponsored in part by the Air Force Cambridge Research Laboratories, Office of Aerospace Research, under contract AF19(628)-419, and by the Rome Air Development Center, under contract AF30(602)-3342.

ABSTRACT

This paper defines a programming language system called AMBIT (Algebraic Manipulation By Identity Translation). The AMBIT language is intended for the precise description of the operations of mathematics in general, and programs in the languages are suitable for efficient compilation and execution by an automatic computer. The language is distinguished by its adherence to an important portion of the conventional notation of algebra, the "identity". AMBIT uses the "identity" to express in a single linguistic structure an arbitrarily complex sequence of elementary symbol-manipulation operations, just as FORTRAN and ALGOL use the "formula" to express in a single linguistic structure an arbitrarily complex sequence of arithmetic operations. Thus AMBIT attempts to serve algebra as FORTRAN and ALGOL serve arithmetic.

CONTENTS

1. INTRODUCTION.

The AMBIT programming language arose from the decision to base the design of a programming language for mathematical symbol manipulation on the conventional notation for the identity. Given this objective, the design of a suitable programming language involved three major tasks: the definition of a suitable representation for the data on which a program operates, the definition of an algorithmic interpretation of the identity, and the definition of a suitable program logic to control the execution of the identities. These three areas of design are closely related, and the design of the programming language required extensive experimentation with the various options available. This paper is a report of this experimentation; it consists primarily of the definition of the programming system which was the result of the experimentation, but also includes, where practical, a discussion of the criteria which dictated the final design.

The AMBIT system is designed for implementation on a computer and is intended for the automatic performance of mathematical algorithms; but the computer-oriented aspect of AMBIT has been suppressed to an unusual degree. With the exception of a brief section on the computer implementation of AMBIT, this paper will discuss AMBIT without reference to automatic computers. The AMBIT programming language will be regarded as a language for the communication of mathematical algorithms between human mathematicians, and the AMBIT system will be regarded as a system for the execution of mathematical algorithms by a human mathematician. The manual execution of an AMBIT program is a practical undertaking for many non-trivial examples; it is practical because a human mathematician regards the application of an identity to mathematical data as a simple and natural action, even though the action performed may, in fact, consist of a sequence of many elementary operations of symbol manipulation.

The area of application of the AMBIT system is not limited to elementary algebra; and it is most certainly not limited to such simple algorithms as the "multiplying-out" of an equation which has been chosen as the example for discussion in this paper. The design of AMBIT was based largely on the

experimental programming of a selected set of algorithms. These algorithms were programmed and re-programmed as changes were made in the design of the language, were manually executed for typical data, and were used as a practical test of the facilities included in the language. A brief mention of the programs involved in these experiments will suggest the scope of AMBIT. Programs drawn from elementary algebra include programs for the "clearing of fractions" from an equation, the "multiplying-out" of an equation, the substitution of an expression for a variable in an equation, and the canonical ordering of factors and terms of an equation (as a preliminary to simplification of the equation). Programs drawn from other areas of classical mathematics include a program for formal differentiation, programs for the union and intersection of sets and a program which tests a proposition of the predicate calculus for theoremhood. Programs for general symbol manipulation include three programs for syntactic analysis (each embodying a different technique) and a program for non-trivial tree manipulation.

2-128

2-129

The AMBIT programming language resembles certain existing programming languages for symbol manipulation in varying degree, to the extent that these languages have approached the utilization of an algorithmic interpretation of the identity [1, 2, 3, 4]. The AMBIT language is distinguished from these languages by the fact that AMBIT is entirely and explicitly based on a notation for the identity and uses the identity alone to specify virtually every test and modification of the data on which an AMBIT program operates.

1.1. An Example from Conventional Algebra.

Consider, first, an example of algebraic symbol manipulation as it might be performed by a human mathematician using conventional methods and conventional notation. Suppose the mathematician is given the equation

$$(\alpha+2.5)\times(y^3-m)=1 \qquad \ldots\text{Eq. (1)}$$

and is asked to produce an algebraically equivalent equation which is "multiplied-out". The mathematician will quickly produce a result such as

$$(\alpha \times y^3 + 2.5 \times y^3) - (\alpha \times m + 2.5 \times m) = 1 \qquad \ldots \text{Eq.}(2)$$

Suppose, however, that this result is challenged; that is, the mathematician is asked to "prove" that eq. (2) is algebraically equivalent to eq. (1). Then the mathematician might write down the identities

$$A \times (B \pm C) \equiv A \times B \pm A \times C \qquad \ldots \text{Eq.}(3)$$
$$(A \pm B) \times C \equiv A \times C \pm B \times C \qquad \ldots \text{Eq.}(4)$$

and then write the following derivation of eq. (2) from eq. (1):

1.	$(\alpha + 2.5) \times (y^3 - m) = 1$	Given
2.	$(\alpha + 2.5) \times y^3 - (\alpha + 2.5) \times m = 1$	Line 1 and Eq.(3)
3.	$(\alpha \times y^3 + 2.5 \times y^3) - (\alpha + 2.5) \times m = 1$	Line 2 and Eq.(4)
4.	$(\alpha \times y^3 + 2.5 \times y^3) - (\alpha \times m + 2.5 \times m) = 1$	Line 3 and Eq.(4)

This derivation is, strictly speaking, merely the outline of a formal proof, a proof which would make direct use of the set of axioms for the algebra involved. However, the derivation does make clear the assumptions made (the identities) and the steps performed (the successive lines of the derivation) in obtaining eq. (2) from eq. (1).

The derivation just given suggests an algorithm for the "multiply-out" operation. Let the "forward application" of an identity be an application in which the symbol '≡' is interpreted as "may be replaced by" rather than "may replace". Then an algorithm for "multiplying-out" a given equation may be defined as follows:

1. Accept the given equation as the "current equation".
2. Attempt to modify the current equation by a forward application of any one of the relevant identities (that is, eqs. 3 and 4) to any expression in the current equation. If the attempt succeeds, then repeat this step. Otherwise, continue to Step 3.
3. Accept the current equation as the desired result.

The simplicity of the algorithm just given arises from the fact that the "relevant identities" referred to in Step 2 embody in themselves an important

part of the strategy for the multiply-out procedure. That is, any application of any relevant identity brings the current equation closer to the desired result; and the failure of all relevant identities to apply to the current equation is a sufficient condition for the completion of the procedure. Thus an identity not only specifies a change in the current equation, but also controls, by its success or failure, the logical flow of the algorithm.

1.2. The AMBIT System.

In order to define the AMBIT programming language, it will be useful to postulate an AMBIT "system" which contains the AMBIT programming language as one of its parts. The AMBIT system consists of three parts, as follows:

1. A data language for the formal representation of citations from algebra. The data language is a set of symbol sequences, each of which is called a "data string".
2. A programming language for the formal representation of algorithms for the manipulation of citations from algebra. The programming language is a set of symbol sequences, each of which is called an "AMBIT program".
3. A program executer to modify a given data string under the guidance of a given AMBIT program. The program executer (which may be a human mathematician or an automatic computer) is constrained to follow certain well-defined "execution rules".

The central feature of the AMBIT programming language is the "replacement rule", which corresponds to the "identity" of conventional mathematical notation. The replacement rule is defined so that, on the one hand, it closely resembles in appearance and function the conventional identity and, on the other hand, it has a sufficiently precise and simple interpretation to be efficiently interpreted by an automatic computer. The replacement rule is the principal device by which the modification of a given citation from algebra is specified. The remainder of the AMBIT programming language is designed to

supply a suitable "algorithmic environment" for replacement rules. This algorithmic environment permits the programmer to control the order in which replacement rules are applied to a given citation from algebra, and to specify unambiguously that portion of the citation to which a given replacement rule is to be applied.

An "AMBIT programmer" is a programmer who makes use of the AMBIT system for the automatic performance of algebraic symbol manipulation. Initially, the AMBIT programmer has an informal statement of the operation which is to be performed (such as the phrase, "multiply-out the given equation"), a collection of identities relevant to the operation (such as Eqs. (3) and (4), above), and a citation from algebra to which this operation is to be applied (such as Eq. (1), above). The use of the AMBIT system by the programmer then proceeds as follows:

1. The programmer writes an AMBIT program which incorporates the given identities into a context in which they have a formally defined meaning and in which the "when and where" of their application to the given citation is correctly specified.
2. The programmer makes minor changes necessary to convert the given citation into an AMBIT data string.
3. The programmer submits the AMBIT program and the AMBIT data string to an AMBIT program executer.
4. The AMBIT program executer modifies the data string under the guidance of the AMBIT program and returns the program and the modified data string to the programmer.
5. The programmer makes the minor changes necessary to convert the modified data string into a conventional citation from algebra and accepts this citation as his result.

The example considered in this paper will necessarily be a simple one. In general, however, the data string is not restricted to a single equation, but rather may be any collection of mathematical information. Similarly, the AMBIT program is not restricted to a simple operation, but may be a complex of simple

operations, each of which is applied to an appropriate part of the data string at an appropriate time.

Section 2 of this paper will examine the execution of an example AMBIT program in detail. The AMBIT program used in the example will be a formally correct program, but the discussion of the program is intended to invoke an intuitive understanding of the program rather than an understanding of the detailed mechanics of the AMBIT system. Section 3 is a complete formal definition of the AMBIT system. Section 4 is a discussion of the motivation behind the important aspects of the formal definition of the AMBIT system, and touches very briefly upon the intended computer implementation of the AMBIT system. Section 5 concludes the paper with a discussion of the proposed extensions of the system defined in this paper.

2. AN EXAMPLE OF AMBIT.

The AMBIT system is a formally defined and computer-implementable system; but the language used in the AMBIT system resembles the familiar language of conventional algebra. It is therefore appropriate to begin the discussion of the AMBIT system with an informal discussion of the execution of an example AMBIT program and to defer the formal definition of the AMBIT system to a later section of this paper. Any conflict between the informal discussion given here and the formal definition of AMBIT given later should, of course, be resolved in favor of the latter. The AMBIT program discussed here performs a very simple task ("multiplying-out" an equation) and is, in fact, an unimaginative programming of that task. The program is appropriate for a detailed introductory study, but it is not a good example of the power of the AMBIT system.

2.1. The Data String.

In this example of the execution of an AMBIT program, the given data string will be

$$EQ1\Delta(((alpha+2.5)\times((y\uparrow 3)-m))=1)$$

and the result data string will be

```
EQ2Δ((((alphax(y↑3))+(2.5×(y↑3)))
     -((alphax×m)+ (2.5×m))) = 1)
```

These two data strings are eqs. (1) and (2) of the Introduction, expressed in a notation acceptable to the AMBIT system. The conversion of a conventional mathematical citation into an AMBIT data string consists of three steps, as follows:

a. Certain symbols (such as 'α') and certain arrangements of symbols (such as 'y^3') which are not available in the AMBIT system are replaced by suitable equivalents (such as 'alpha' and 'y↑3').

b. Equation labels (such as '...Eq. (1)' and '...Eq. (2)') and similar unique identifiers are expressed as AMBIT "pointers" (such as 'EQ1Δ' and 'EQ2Δ') and are moved to the beginning of the symbol sequences they label.

c. Mathematical citations are fully parenthesized.

Step (a) is a familiar requirement of computer-implementable systems. Step (b) is a small matter. Step (c), however, is an unreasonable demand to make on the user of the AMBIT system. The full parenthesization of mathematical citations is essential to the efficient processing of the citations by the AMBIT system; but sub-programs can be written in the AMBIT programming language for the insertion of (and the deletion of) redundant parentheses. The discussion of such sub-programs exceeds the limited scope of the example considered here, and therefore it has been assumed that the data string is input (and output) in fully parenthesized form. In actual practice, the AMBIT programmer would have access to standard sub-programs (or sub-routines), written in AMBIT, for the parenthesization of input data and the de-parenthesization of output data. Thus an AMBIT program would begin by fully parenthesizing the input data string, would then perform the specified mathematical manipulations, and would conclude by removing redundant parentheses from the result data string.

The AMBIT system includes a large vocabulary of formally defined English words, called "data types", which are used to denote those sub-sequences

which frequently occur in a data string which is a citation from algebra. For the example considered here, the following informal definition of five of these data types will suffice:

a. pointer. This data type denotes an identifier followed by 'Δ'. (An identifier is a letter followed by a sequence of letters and digits.) The given data string above contains one pointer, namely 'EQ1Δ'.

b. element. This data type denotes a number, a variable, an algebraic operator, or a pointer. There are 12 elements in the given data string above, namely 'EQ1Δ', 'alpha', '+', '2.5', '×', 'y', '↑', '3', '-', 'm', '=', and '1'.

c. phrase. This data type denotes an element or a parenthesized symbol sequence. There are 17 phrases in the given data string, namely the 12 elements just listed and '(alpha+2.5)', '(y↑3)', '((y↑3)-m)', '((alpha+2.5)×((y↑3)-m))', and '(((alpha+2.5)×((y↑3)-m)) = 1)'.

d. segment. This data type denotes an element or a single parenthesis. The given data string consists of 22 segments, namely, 'EQ1Δ', '(', '(', 'alpha', '+', etc.

e. sign. This data type denotes the symbol '+' or '-'.

2.2 The Program.

In this example of the execution of an AMBIT program, the program executed will be

```
1.  begin   phrase dummy A, B, C, Q;
2.          segment dummy seg;
3.          sign dummy sign;
4.          EQ1Δ Q → EQ2Δ Q ;
5.  SCAN:   EQ2Δ Q → EQ2Δ pΔ Q ;
6.  MULT:   if pΔ Ax(B sign C) → (AxB) sign (AxC)
7.          or pΔ (A sign B)xC → (AxC) sign (BxC)
8.          then go to SCAN;
9.          pΔ seg → seg pΔ ;
10.         if EQ2Δ Q pΔ → EQ2Δ Q
11.         then go to EXIT
12.         else go to MULT ;
13. EXIT:   end
```

2-135

2-134

The line numbers which appear on the left are not a part of the program; they have been inserted to facilitate this discussion of the program. The example above is in all other respects a formally correct AMBIT program.

The program is designed to "multiply-out" a given equation in precisely the sense that this operation was defined in the Introduction. The two identities given in the Introduction (eqs. 3 and 4) have been incorporated in the program (Lines 6 and 7). These identities are embedded in a context which partially defines their interpretation (Lines 1 - 3) and which controls their application to the data string (Lines 4 - 5 and 8 - 13). The given equation must appear in the data string as a fully parenthesized symbol sequence which is preceded by the pointer 'EQ1Δ'. The data string may consist of much more than this particular equation (for example, other equations preceded by other pointers); but only that portion of the data string which is the equation preceded by 'EQ1Δ' will be affected by the execution of the program.

Lines 1 - 3 of the example program are the "declarative part" of the program. The declarative part does not cause the program executer (PE) to take any action on the data string; rather, it provides the PE with information which is necessary for the correct interpretation of the "imperative part", Lines 4 - 13, of the program. Line 1 informs the PE that the identifiers 'A', 'B', 'C', and 'Q' are "phrase dummy-variables", and that any appearance of any one of these identifiers in the imperative part of the program is not to be interpreted literally but rather as a designation for an arbitrary phrase in the data string. Similarly, Lines 2 and 3 inform the PE that 'seg' and 'sign' are designations for an arbitrary segment and an arbitrary sign, respectively. (The words "phrase", "segment", and "sign" are used here in the special sense defined in Sec. (2.1), above.)

The imperative part of the program consists primarily of "replacement rules". A replacement rule consists of two "string descriptions" separated by the symbol '→'. The string description to the left of '→' is the "citation", and that to the right is the "replacement". A replacement rule instructs the PE to find a sub-sequence of the current data string which is described by the citation and to replace that sub-sequence by a symbol sequence which is described by the replacement. For example, the replacement rule on Line 5 of the example program is interpreted as follows:

> Find a sub-sequence of the current data string which **satisfies** the following description: 'EQ2Δ', followed by a symbol sequence which is an arbitrary phrase and which is hereby named 'Q'. Replace the symbol sequence just found with a symbol sequence which satisfies the following description: 'EQ2Δ', followed by 'pΔ', followed by a symbol sequence which is identical to the symbol sequence previously named 'Q'.

Note that pointers (EQ1Δ, EQ2Δ, and pΔ in this program) are always interpreted literally in the string description.

The modification of the current data string specified by a replacement rule may or may not be possible, according as the citation describes a sub-sequence which currently does or does not exist in the data string. If the modi-

fication is possible, then the PE performs it and the replacement rule is said to "succeed"; otherwise, the PE leaves the data string unmodified and the execution of the replacement rule is said to "fail". The failure of a given execution of a replacement rule is not necessarily a program error; on the contrary, the success or failure of a replacement rule is frequently and legitimately used to control the subsequent course of program execution.

The program executer normally executes the replacement rules in the order in which they appear in the program; exceptions to this rule occur in two ways. First, the "control imperatives" (such as 'go to SCAN') have the familiar effect of interrupting sequential execution and sending the PE to the designated program label (such as 'SCAN:'). Second, the logical connectives 'if', 'then', 'else', 'or' (and others which are not illustrated in the program) may cause certain replacement rules to be skipped over. The use of these connectives in AMBIT is similar to their use in a conventional English imperative sentence; however, certain popular ambiguities of conventional English are formally resolved in AMBIT. For example, if A and B are two actions, and an agent is told, in conventional English, "Do A or B", the agent may well inquire "Which shall I try to do first? And if both A and B are possible, shall I do both? " According to the conventions of AMBIT, the reply is "Always proceed from left to right. Never do more than is strictly necessary. In this case, if both A and B are possible, do A and skip B." The complete interpretation of the program logic of AMBIT is given in Sec. (3), below.

The strategy of the example program is primitive but convincing. The equation label (pointer) 'EQ1Δ' is changed to 'EQ2Δ' in anticipation of a successful outcome (Line 4). The pointer 'pΔ' is inserted to the left of the equation (Line 5). The symbol sequence which immediately follows 'pΔ' is tested to determine whether or not it begins with a sub-sequence to which either of the "multiply-out" identities is applicable (Lines 6 and 7). If an identity is applicable, it is applied (causing the multiplying-out of part of the equation) and 'pΔ' is moved back to the beginning of the equation (Lines 8 and 5). Otherwise, 'pΔ' is advanced to the right one segment in the equation (Line 9) and an end

test is made. If 'pΔ' stands at the right end of the equation, then it is removed from the equation (Line 10) and exit from the program occurs; otherwise, an attempt is made to apply the "multiply-out" identities at the new position of 'pΔ'.

2.3. The Execution of the Program.

An annotated trace of the execution of Program 1 on the example given data string will follow. Several abbreviations are used to shorten the statement of the trace. "Line i succeeds, giving" means "the replacement rule on Line i is attempted and succeeds, giving the following as the current data string:". "Line i fails" means "the replacement rule on Line i is attempted and fails, leaving the current data string unchanged." "Line i is ignored" means "the replacement rule or control imperative on Line i is skipped over in compliance with the relevant program logic."

1. At the beginning of program execution, the data string is
   EQ1Δ(((alpha+2.5)×((y↑3)-m)) = 1)
   The PE begins reading the program at Line 1. Lines 1 - 3 cause no modification of the data string.
2. Line 4 succeeds, giving
   EQ2Δ(((alpha+2.5)×((y↑3)-m)) = 1)
3. Line 5 succeeds, giving
   EQ2Δ pΔ(((alpha+2.5)×((y↑3)-m)) = 1)
4. Lines 6 and 7 fail. Line 8 is ignored. Line 9 succeeds (for seg = '('), giving
   EQ2Δ(pΔ((alpha+2.5)×((y↑3)-m)) = 1)
   Line 10 fails. Line 11 is ignored. Line 12 sends the PE to Line 6.
5. Lines 6 and 7 fail. Line 8 is ignored. Line 9 succeeds (for seg = '(') giving
   EQ2Δ((pΔ(alpha+2.5)×((y↑3)-m)) = 1)
   Line 10 fails. Line 11 is ignored. Line 12 sends the PE to Line 6.

6. Line 6 succeeds (for A = '(alpha+2.5)', B = '(y↑3)', sign = '-', and C = 'm'), giving

```
EQ2Δ((((alpha+2.5)×(y↑3))
      -((alpha+2.5)×m)) = 1)
```

Line 7 is ignored. Line 8 sends the PE to Line 5.

7. Line 5 succeeds, giving

```
EQ2Δ  pΔ((((alpha+2.5)×(y↑3))
        -((alpha+2.5)×m)) = 1)
```

8. The following loop is executed three times:

Lines 6 and 7 fail. Line 8 is ignored. Line 9 succeeds (moving 'pΔ' one segment to the right). Line 10 fails. Line 11 is ignored. Line 12 sends the PE to Line 6.

After the third execution, the data string is

```
EQ2Δ(((pΔ(alpha+2.5)×(y↑3))
        -((alpha+2.5)×m)) = 1)
```

2-139

2-138

9. Line 6 fails. Line 7 succeeds (for A = 'alpha', sign = '+', B = '2.5', C = '(y↑3)'), giving

```
EQ2Δ((((alpha×(y↑3))+(2.5×(y↑3)))
      -((alpha+2.5)×m)) = 1)
```

Line 8 sends the PE to Line 5.

10. Line 5 succeeds ('pΔ' is inserted immediately after 'EQ2Δ'). The loop described in Step 8, above, is executed 25 times (moving 'pΔ' 25 segments to the right). After the 25-th execution, the data string is

```
EQ2Δ((((alpha×(y↑3))+(2.5×(y↑3)))
    -(pΔ (alpha+2.5)×m)) = 1)
```

and the PE is about to read Line 6.

11. Line 6 fails. Line 7 succeeds (for A = 'alpha', sign = '+', B = '2.5', and C = 'm'), giving

```
EQ2Δ((((alpha×(y↑3))+(2.5×(y↑3)))
      -((alpha×m)+(2.5×m)))= 1)
```

Line 8 sends the PE to Line 5.

12. Line 5 succeeds ('pΔ' is inserted immediately after 'EQ2Δ'). The loop described in Step 8, above, is executed 40 times. After the 40-th execution, the data string is

EQ2Δ((((alpha×(y↑3))+(2.5×(y↑3)))
-((alpha×m)+(2.5×m))) = 1 pΔ)

and the PE is about to read Line 6.

13. Lines 6 and 7 fail. Line 8 is ignored. Line 9 succeeds (moving 'pΔ' to the right of the last parenthesis in the equation). Line 10 succeeds, giving

EQ2Δ((((alpha×(y↑3))+(2.5×(y↑3)))
-((alpha×m)+(2.5×m))) = 1)

Line 11 sends the PE to the end of the program (Line 13). The data string is the multiplied-out equivalent of the input data string shown in Step 1 above. The program executer returns the AMBIT program (unchanged) and the data string (multiplied-out) to the AMBIT programmer and expires.

3. A FORMAL DEFINITION OF THE AMBIT SYSTEM.

This section will include formal definitions of the three parts of the AMBIT system: the data language, the programming language, and the program executer. The definition of the data language (that is, the set of symbol sequences each of which is a data string) is given by means of "data-language formulae" (DF) and "data-language notes" (DN). Similarly, the definition of the programming language (that is, the set of symbol sequences each of which is an AMBIT program) is given by means of "programming-language formulae" (PF) and "programming-language notes" (PN). Finally, the definition of the program executer (that is, the agent which modifies a given data string under the guidance of a given AMBIT program) is given by means of "execution rules" (ER).

The DF and the PF are "metalinguistic formulae", written in a notation which is defined in the revised ALGOL 60 report [5] and which has been used in the definition of the syntax of ALGOL 60 and several other programming languages. The DN and the PN are definitions and restrictions which are expressed in conventional English; they are essential because a set of metalinguistic formulae

alone is not adequate for the complete definition of either the data language or the programming language. The ER are written in a notation which closely resembles the "descriptive language for symbol manipulation" introduced by Floyd in [1].

3.1. The Data Language.

The definition of the data language given by the DF and DN below can be summarized informally as follows:

a. A data string is a sequence of "data symbols". The set of data symbols consists of the letters (upper and lower case), the digits, the usual arithmetic, relational, and logical operators (drawn from ALGOL 60), the parentheses, the blank, and the eight special symbols which follow:

true false . $_{10}$ $\oplus$ $\ominus$? Δ

b. Parentheses may appear in the data string only in ordered nested pairs, in accordance with the conventional use of parentheses in mathematics.

c. Certain sub-sequences of a data string, called "pointers", must be unique in the data string.

The DF and DN below define the data language consistently with the informal definition just given; but this is not their only function. The data formulae also define supplementary metavariables, or "data types", which are useful in the description of sub-sequences of a given data string. The definition of these supplementary metavariables is not essential to the definition of the data language itself. For example, the definition of "mark" given by DF.6 through DF.9 could be given as a single metalinguistic formula at the expense of the omission of the definitions of the supplementary metavariables "logical", "relational", and "arithmetic". Furthermore, DF.17 through DF.32 make no contribution whatever to the definition of the data language and their sole function is the definition of supplementary metavariables. While the supplementary metavariables are not essential to the definition of the data language, they are essential to the formal definition of the AMBIT system as a whole.

Specifically, the AMBIT program executer (see Sec. (3.3)) is expected to know and make use of the definitions of these metavariables in executing an AMBIT program.

Note that the "blank" symbol is explicitly mentioned in the metalinguistic formulae below and is formally defined by DN.5. Blanks may not appear in the data language except as explicitly allowed by the DF. Thus, for example, an "identifier" (as defined by DF.16) must not contain a blank.

DF: Metalinguistic Formulae for the Data String.

1. ⟨ data string⟩ :: = ⟨ blank⟩ ⟨ string⟩ ⟨ blank⟩
2. ⟨ string⟩ :: = ⟨ phrase⟩ ⟨ string⟩ | ⟨ blank⟩ ⟨ string⟩ | ⟨ empty sequence⟩
3. ⟨ phrase⟩ :: = ⟨ parenthesized string⟩ | ⟨ element⟩
4. ⟨ parenthesized string⟩ :: = (⟨ string⟩)
5. ⟨ element⟩ :: = ⟨ mark⟩ | ⟨ word⟩
6. ⟨ mark⟩ :: = ⟨ logical⟩ | ⟨ relational⟩ | ⟨ arithmetic⟩
7. ⟨ logical⟩ :: = ≡ | ⊃ | ∨ | ∧ | ¬
8. ⟨ relational⟩ :: = < | ≤ | = | ≥ | > | ≠
9. ⟨ arithmetic⟩ :: = + | - | × | / | ↑
10. ⟨ word⟩ :: = ⟨ alphanumeric⟩ ⟨ word⟩ | ⟨ alphanumeric⟩
11. ⟨ alphanumeric⟩ :: = ⟨ letter or digit⟩ | true | false | . | $_{10}$ | ⊕ | ⊖ | ? | Δ
12. ⟨ letter or digit⟩ :: = ⟨ letter⟩ | ⟨ digit⟩
13. ⟨ letter⟩ :: = a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
14. ⟨ digit⟩ :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
15. ⟨ pointer⟩ :: = ⟨ identifier ⟩ Δ | Δ
16. ⟨ identifier⟩ :: = ⟨ identifier⟩ ⟨ letter or digit⟩ | ⟨ letter⟩
17. ⟨ value⟩ :: = ⟨ number⟩ | ⟨ Boolean⟩
18. ⟨ number⟩ :: = ⟨ real⟩ | ⟨ integer⟩

19. ⟨ real⟩ :: = ⟨ sign⟩ ⟨ unsigned real⟩ | ⟨ unsigned real⟩

20. ⟨ integer⟩ :: = ⟨ sign⟩ ⟨ unsigned integer⟩ | ⟨ unsigned integer⟩

21. ⟨ Boolean⟩ :: = true | false

22. ⟨ unsigned real⟩ :: = ⟨ decimal⟩$_{10}$⟨ exponent⟩ | ⟨ decimal⟩ |
⟨ unsigned integer⟩$_{10}$ ⟨ exponent⟩ | $_{10}$⟨ exponent⟩

23. ⟨ decimal⟩ :: = ⟨ unsigned integer⟩ . ⟨ unsigned integer⟩ |
⟨ unsigned integer⟩ . | .⟨ unsigned integer⟩

24. ⟨ exponent⟩ :: = ⟨ exponent sign⟩ ⟨ unsigned integer⟩ | ⟨ unsigned integer⟩

25. ⟨ unsigned integer⟩ :: = ⟨ digit⟩ ⟨ unsigned integer⟩ | ⟨ digit⟩

26. ⟨ exponent sign⟩ :: = ⊕ | ⊖

27. ⟨ sign⟩ :: = + | -

28. ⟨ binary logical⟩ :: = ≡ | ⊃ | ∨ | ∧

29. ⟨ segment⟩ :: = ⟨ element⟩ | ⟨ parenthesis⟩

30. ⟨ parenthesis⟩ :: = (|)

31. ⟨ character⟩ :: = ⟨ mark⟩ | ⟨ alphanumeric⟩

32. ⟨ data symbol⟩ :: = ⟨ character⟩ | ⟨ parenthesis⟩ | ⟨ blank⟩

2-143

DN: Notes on the Data String

1. The context of the data string (Restriction). A symbol sequence is a data string only if it appears in a context in which the reader or the AMBIT program executer expects to find a data string. This is necessarily an informal restriction since the context of the data string is, in general, external to the AMBIT system.

2. Complete instances (Definition).

Let M be any metavariable defined by the DF or the PF. An instance, s, of a symbol sequence is a "complete instance of an M" if and only if

a. s is an instance of an M, and

b. s appears in a context in which it is not immediately preceded by a symbol sequence x such that xs is an M, and

c. s appears in a context in which it is not immediately followed by a symbol sequence y such that sy is an M.

Example. Consider the following symbol sequence:

sin x = (2.83+c)

In this symbol sequence, 'sin', 'x', '=', '2.83', '+', and 'c' are the only complete elements. The only complete segments are the complete elements just named plus '(' and ')'. The only complete phrases are the complete elements just named and '(2.83+c)'. The only complete strings in the symbol sequence are '2.83+c' (N.B.) and the entire symbol sequence. The sub-sequence '83' is a complete integer; it is also a number, but it is not a complete number. The sub-sequence 'si' is an identifier and an element, but it is not a complete identifer or a complete element.

3. Separation of pointers (Restriction). A complete instance of a pointer must not appear in the data string unless it is a complete instance of a word.

4. Uniqueness of pointers (Restriction). Two or more complete instances of the same pointer must not be contained in the same data string.
Example. Consider the three symbol sequences which follow:

sin x = (2.83pΔ+c)
sin x = pΔ(2.83 pΔ+c)
sin x = (2.83 pΔ+c)

The first two symbol sequences are not legal data strings because they violate DN.3 and DN.4, respectively. The third is a legal data string.

5. Special metavariables (Definition). Two of the metavariables which appear in the DF, "empty sequence" and "blank", are not defined by the DF; they are defined here.

a. Empty sequence. An empty sequence is a portion of a printed line between two immediately adjacent symbols which occupies no space in the printed line. (Thus any number of empty sequences may be imagined to exist between two adjacent symbols.) An empty sequence is not a symbol, but (by definition) it is a symbol sequence.

b. Blank. A blank is an unmarked portion of a printed line which appears between two marked portions of that printed line, provided that the unmarked portion is wide enough to be clearly distinguished from the interstice which, in all printed documents, appears between any pair of "immediately adjacent" printed symbols. Further, wherever a symbol sequence is interrupted at the end of a printed line and continued at the beginning of the next printed line (because of practical limitations on the length of a single printed line), that interruption is interpreted as a blank.

6. The identity of symbol sequences (Definition). Two symbol sequences are identical if they consist of respectively identical symbols. Two non-blank symbols are identical if, within the imperfections of the printing mechanism in use, they are physically identical in shape and orientation. Two blank symbols are identical in every case, even though they may differ in physical width or one of them may arise from an end-of-line condition; thus the representation of blanks is not formally controlled in the AMBIT system.

7. The value of an identifier (Definition). The "value" of an identifier is defined with respect to a particular data string, such as the data string on which a given AMBIT program is being executed. Let ID be any identifier, and let PTR be the pointer which is formed by appending 'Δ' to ID. If there exists a sub-sequence of the data string which consists of a complete instance of the pointer PTR followed by a blank followed by a complete instance of an arbitrary phrase, PHR, then the phrase PHR is the "value" of the identifier ID. If no such sub-sequence exists in the data string, then the data symbol '? ' is the "value" of the identifier ID.
Example. Consider the following data string:

pΔ (a+qΔ 2.83 rΔ) sΔ ? "

With respect to this data string, the value of 'p' is '(a + qΔ 2.83 rΔ)', the value of 'q' is '2.83', and the value of 'r', 's', and 't' is '? '.

8. Editing operations (Definition). The four editing operations "expand", "pack", "space fully", and "space legibly" are defined here. Each of these editing operations modifies the symbol sequence to which it is applied by the insertion or deletion of blanks.

a. expand. For every pair of immediately adjacent symbols which is such that both symbols are alphanumerics, insert a blank between the two symbols unless one (or both) of the symbols is contained in a complete instance of a pointer.

b. pack. For every pair of symbols separated by a blank which is such that both symbols are alphanumerics, delete the blank between the two symbols unless one (or both) of the symbols is contained in a complete instance of a pointer.

c. space fully. For every pair of immediately adjacent, non-blank symbols such that at least one of the symbols is not an alphanumeric, insert a blank between the two symbols.

d. space legibly. For every pair of symbols separated by a blank which is such that at least one of the symbols is not an alphanumeric, delete the blank between the two symbols provided that this deletion increases the legibility of the symbol sequence. (The concept of "legibility" is left undefined).

2-146

A symbol sequence is said to be "expanded", "packed", "fully spaced", or "legibly spaced" if the application of the corresponding operation to the symbol sequence does not result in the insertion or deletion of any blanks.

Example. Consider the following symbol sequence:

sin x = (plΔ 2.83 + c)

Each of the four symbol sequences which follow is produced by the application of one of the editing operations, as indicated, to the symbol sequence just given:

```
s i n x = ( plΔ 2 . 8 3 + c )        [expand]
sinx = ( plΔ 2.83 + c )              [pack]
sin x = ( plΔ 2.83 + c )             [space fully]
sin x = ( plΔ 2.83+c)                [space legibly]
```

Since the application of "space fully" caused no change in the given symbol sequence, the given symbol sequence was "fully spaced".

9. The evaluate operation (Definition). The "evaluate" operation replaces a symbol sequence by a symbol sequence which is as fully evaluated as possible with respect to the arithmetic and Boolean operations defined in ALGOL 60. This operation consists of the following steps:

a. If the symbol sequence is not a parenthesized string, then enclose it in parentheses.

b. Replace every '⊕' or '⊖' which immediately follows a '$_{10}$' by '+' or '-', respectively.

c. If Secs. (3.3) and (3.4) of the ALGOL 60 report [5] define an ALGOL value (Boolean, integer, or real) for a parenthesized string contained in the symbol sequence, replace that parenthesized string with the ALGOL value enclosed in parentheses. If the ALGOL value is integer or real, it must be written so that it begins with a sign and does not have leading zeroes; however, the number of digits in the fractional part of a real value (the "precision" of the value) is not specified here (or in the ALGOL report), and may be chosen freely. Apply this step repeatedly until it is no longer applicable.

d. Replace every instance of '+' or '-' which immediately follows an instance of '$_{10}$' by an instance of '⊕' or '⊖', respectively.

Example. Consider the three following symbol sequences:

$((4+16)\times((.125+.375)\uparrow 2))$

$((a+16)\times((.125+.375)\uparrow 2))$

$((\underline{\text{true}} \vee \underline{\text{false}}) \wedge \neg 5\times 3 = 10)$

The result of applying the "evaluate" operation to each of these symbol sequences is, respectively,

(5.0)
((a+16)×(.25))
(true)

3.2. The Programming Language.

The definition of the programming language is divided between metalinguistic formulae (PF) and informal notes (PN), just as was the definition of the data language. The metalinguistic formulae make use of three metavariables which have already been defined; namely, "string" (defined by DF.2), "identifier" (defined by DF.16), and "empty sequence" (defined by DN.5). The informal notes refer to certain concepts already defined, such as that of a "complete instance" (defined by DN.2).

The policy with respect to blanks is more liberal in the programming language, and is not explicitly indicated in the PF. A blank may be inserted or deleted anywhere in an AMBIT program except between two alphanumerics without changing the syntax or semantics of that program. The only important role of the blank is in the "string description", where it is used to denote a blank in the data string.

Four of the PN (PN.2 - PN.5) define the declaration mechanism of AMBIT, which is similar in many respects to that of ALGOL. Two of the PN (PN.6 and PN.7) define the use of program labels, also similar to that of ALGOL. The remaining PN define less familiar aspects of the programming language.

PF: Metalinguistic Formulae for the Program.

1. ⟨ program⟩ :: = ⟨ block⟩
2. ⟨ block⟩ :: = begin ⟨ declarative list⟩ ⟨ imperative list⟩ end
3. ⟨ declarative list⟩ :: = ⟨ declarative statement⟩ ⟨ declarative list⟩ |
⟨ empty sequence⟩

4. ⟨ declarative statement⟩ :: = ⟨ declarator⟩ ⟨ identifier list⟩

5. ⟨ declarator⟩ :: = ⟨ data type⟩ dummy | name | label | literal

6. ⟨ data type⟩ :: =
string | phrase | parenthesized string | element | mark |
logical | relational | arithmetic | word | alphanumeric |
letter or digit | letter | digit | pointer | identifier |
value | number | real | integer | Boolean | unsigned real |
unsigned integer | exponent sign | sign | binary logical |
segment | character

7. ⟨ identifier list⟩ :: = ⟨ identifier⟩ , ⟨ identifier list⟩ | ⟨ identifier⟩

8. ⟨ imperative list⟩ :: = ⟨ attached label⟩ ⟨ imperative list⟩ |
⟨ imperative statement⟩ ⟨ imperative list⟩ | ⟨ empty sequence⟩

9. ⟨ attached label⟩ :: = ⟨ identifier⟩ :

10. ⟨ imperative statement⟩ :: = ⟨ imperative⟩;

11. ⟨ imperative⟩ :: = try ⟨ implication⟩ | ⟨ implication⟩

12. ⟨ implication⟩ :: =
if ⟨ disjunction⟩ then ⟨ disjunction⟩ else ⟨ disjunction⟩ |
if ⟨ disjunction⟩ then ⟨ disjunction⟩ | ⟨ disjunction⟩

13. ⟨ disjunction⟩ :: = ⟨ conjunction⟩ or ⟨ disjunction⟩ | ⟨ conjunction⟩

14. ⟨ conjunction⟩ :: = ⟨ negation⟩ and ⟨ conjunction⟩ | ⟨ negation⟩

15. ⟨ negation⟩ :: = not ⟨ primary imperative⟩ | ⟨ primary imperative⟩

16. ⟨ primary imperative⟩ :: =
⟨ block⟩ | ((imperative⟩) | ⟨ simple imperative⟩

17. ⟨ simple imperative⟩ :: = ⟨ control imperative⟩ |
⟨ special procedure call⟩ | ⟨ replacement rule⟩

18. ⟨ control imperative⟩ :: = go to ⟨ identifier⟩

19. ⟨ special procedure call⟩ :: = ⟨ special procedure name⟩ ((identifier))

20. ⟨ special procedure name⟩ :: = expand | pack | evaluate

21. ⟨ replacement rule⟩ :: = ⟨ string description⟩ → ⟨ string description⟩

22. ⟨ string description⟩ :: = ⟨ string⟩ | null

23. ⟨ program symbol⟩ :: = ⟨ reserved symbol⟩ | ⟨ data symbol⟩

24. ⟨ reserved symbol⟩ :: = ⟨ data type⟩ | , | ; | : | → | null |
begin | end | dummy | name | label | literal | if | then |
else | or | and | not | go to | pack | expand | evaluate

PN: Notes on the AMBIT Program.

1. The context of the program (Restriction). A symbol sequence is a program only if it appears in a context in which the reader or the program executer expects to find a program.
2. Declaring-instances of identifiers (Definition). Let ID_1 be a complete instance of an identifier ID in a given AMBIT program. Let DOR denote some declarator.

   a. If ID_1 is contained in a declarative which begins with a complete instance of the declarator DOR, then ID_1 is a "declaring instance" of ID such that "ID_1 is declared DOR".
   b. If ID_1 is contained in an attached label, then ID_1 is a "declaring instance" of ID such that "ID_1 is declared 'label'".
   c. If ID_1 is contained in a pointer, then ID_1 is a "declaring instance" of ID such that "ID_1 is declared 'name'".
   d. In no other context is ID_1 a "declaring instance" of ID.

   Note: In this and subsequent PN, mnemonic sequences of capital letters, such as 'ID', 'DOR', etc., are used to denote sub-sequences of an AMBIT program, such as identifiers, declarators, etc.
   Example. In the example program in Sec. (2), the instances of 'A', 'B', 'C', and 'Q' on Line 1 are "declaring instances" of these identifiers, and are all "declared 'phrase dummy'". The instance of 'seg' on Line 2 and the instance of 'sign' on Line 3 are "declaring instances" and are "declared 'segment dummy'" and "declared 'sign dummy'", respectively. The instances of 'SCAN', 'MULT', and 'EXIT' on Lines 5, 6, and 13 are "declaring instances" and are "declared 'label'". All instances of 'EQ1', 'EQ2', and 'p' are "declaring instances" and are "declared 'name'". These are the only "declaring instances" of identifiers in the example program.

2-150

3. Uniqueness of declaration (Restriction). Let ID_1 and ID_2 be two complete instances of the same identifier, ID, such that ID_1 and ID_2 are both declaring instances of ID. By PN.2, ID_1 will be declared DOR_1 for some declarator DOR_1, and ID_2 will be declared DOR_2 for some declarator DOR_2. If ID_1 and ID_2 are in the same AMBIT program, then DOR_1 and DOR_2 must be the same declarator.

4. Declared instances of identifiers (Definition). Let ID_1 and ID_2 be any complete instances of the same identifier, ID, such that ID_1 is, and ID_2 is not, a declaring instance of ID. By PN.2, ID_1 will be declared DOR for some declarator DOR. If ID_2 is contained in the smallest block which contains ID_1, then ID_2 is a "declared instance" of ID such that "ID_2 is declared DOR".

Example. In the example program in Sec. (2), the instances of 'A', 'B', 'C', and 'Q' on Lines 4 - 13 are all "declared instances" of these identifiers and are all "declared '*phrase* *dummy*'". The instances of 'seg' on Line 9 and the instances of 'sign' on Lines 6 - 7 are "declared instances" and are "declared '*segment* *dummy*'" and "declared '*sign* *dummy*'", respectively. The instances of 'SCAN', 'EXIT', and 'MULT' on Lines 8, 11, and 12 are "declared instances" and are "declared '*label*'".

5. Existence of declaration (Restriction). Let ID_1 be a complete instance of an identifier ID, such that ID_1 is not a declaring instance of ID. If ID_1 is contained in an AMBIT program, then ID_1 must be a declared instance of ID. Furthermore,

   a. If ID_1 is contained in a replacement rule, then ID_1 must be declared '*literal*','*name*' or 'DT *dummy*', where DT is any data type.
   b. If ID_1 is contained in a control imperative, then ID_1 must be declared '*label*'.
   c. If ID_1 is contained in a special procedure call, then ID_1 must be declared '*name*'.

6. Uniqueness of labelling (Restriction). Two or more complete instances of the same attached label must not be contained in the same AMBIT program.

7. Existence of labelling (Restriction). If a complete instance of an identifier is contained in a control imperative in an AMBIT program, then a complete instance of the same identifier must be contained in an attached label in the same AMBIT program.

8. Citations and replacements (Definition). A sub-sequence of an AMBIT program is a citation if and only if it is a complete string description and it is contained in a replacement rule to the left of the symbol '→'. A sub-sequence of an AMBIT program is a replacement if and only if it is a complete string description and it is contained in a replacement rule to the right of the symbol '→'.

9. Non-ambiguity of citations (Restriction). Unless a citation consists of the symbol 'null', the citation must be such that the repeated application of the following "underlining operation" would eventually underline the entire citation. Let SEG be any complete segment in the given citation. If any one of the following applies, then underline SEG together with any adjacent blanks:

   a. SEG is a pointer.
   b. SEG is a parenthesis, the "matching parenthesis" of which has previously been underlined.
   c. SEG is not an identifier declared 'string dummy' and SEG is immediately adjacent to a previously underlined portion of the citation.
   d. SEG is an identifier declared 'string dummy' and SEG is both immediately preceded by and immediately followed by previously underlined portions of the citation.

The underlining should be done in such a way that it does not become an integral part of the program and should be removed after the completion of the test.

Examples. Three examples of the application of the underlinging operation to the citation of a replacement rule will be given here. Consider first the citation of the replacement rule on Line 6 of the example program in Sec. (2),

namely,

pΔ Ax(B sign C)

By underlining operation (a), 'pΔ' is underlined; then by seven successive applications of (c) the entire citation is underlined. Thus the citation satisfies PN.9. In fact, for any citation which does not contain an identifier declared 'string dummy', PN.9 reduces to the requirement that the citation either shall be 'null' or shall contain at least one instance of a pointer. Consider next the replacement rule

(S + pΔ P) → (pΔ S + P)

where 'S' and 'P' are assumed to be declared 'string dummy' and 'phrase dummy', respectively. The application of the underlining operations to the citation of this replacement rule proceeds as follows:

2-153

2-152

(S + pΔ P)	by one application of (a)
(S + pΔ P)	by three applications of (c)
(S + pΔ P)	by one application of (b)
(S + pΔ P)	by one application of (d)

Thus the citation satisfies PN.9. Consider finally the replacement rule

S + pΔ P → pΔ S + P

where 'S' and 'P' are declared as before. The citation of this replacement rule does not satisfy PN.9, since no applications of underlining operations will result in the underlining of the identifier 'S' (which has been assumed to be declared 'string dummy'). Thus the replacement rule is not a legal AMBIT structure.

10. Non-ambiguity of replacements (Restriction). Let "C→R" be any replacement rule (where C is a citation and R is a replacement). If a complete instance of an identifier which is declared 'DT dummy' (where DT is any

data type) is contained in R, then at least one complete instance of that identifier must be contained in C.

11. Conservation of matching of parentheses (Restriction). Let "C→R" be any complete replacement rule which contains a complete instance of an identifier declared 'segment dummy' where C is a citation and R is a replacement). Examine the replacement rule ignoring everything which is not a parenthesis or a complete instance of an identifier declared 'segment dummy'. The replacement rule examined in this way must appear to have a replacement which is identical to its citation.
Example. Consider the replacement rule on Line 9 of the example program. When everything but parentheses and complete identifiers declared 'segment dummy' are ignored, this replacement rule rule becomes 'seg→seg'. Thus this replacement rule satisfies PN.11. The purpose of the restriction of PN.11 is to exclude any use of an identifier declared 'segment dummy' to cause an illegal re-arrangement or replication of single parentheses; note that an identifier declared 'segment dummy' is the only type of identifier which can denote an unmatched parenthesis.

12. Conservation of uniqueness of pointers (Restriction). Two or more complete instances of the same pointer or of the same identifier declared 'pointer dummy' must not be contained in the same replacement.

13. Plausibility of citations (Restriction). Two or more complete instances of the same pointer or of the same identifier declared 'pointer dummy' must not be contained in the same citation.

14. Convention for empty string descriptions (Restriction). A citation must not be an empty sequence. A replacement must not be an empty sequence. (The symbol 'null' is used as a string description which denotes an empty sequence in the data string.)

2-154

3.3. The Program Executer.

The program executer (PE) may be a human mathematician or an automatic computer; in either case, the actions taken by the PE are fully defined by the "execution rules" (ER) given in this section. The formalism used to express the ER is less familiar than the formalism used to express the DF and the PF, and is defined in the following paragraphs.

An ER consists of a single "scanning rule" followed by any number (possibly zero) of "conditions" followed by any number (possibly zero) of "actions". The scanning rule is expressed in a formalism which will be discussed below; the conditions and actions are expressed in informal language. Headings and informal declarations appear above sub-sets of the execution rules. A heading indicates the type of sub-sequence of an AMBIT program to which a sub-set of the execution rules applies; for example, ER.2.1 and ER.2.2 appear under the heading "Block" and apply to that type of sub-sequence of an AMBIT program. The informal declarations contribute to the interpretation of the scanning rules.

2-154

2-155

A scanning rule consists of a pair of "diagrams" separated by the symbol '⇒'. A diagram is a symbol sequence composed of "diagram variables", "diagram literals", and blanks. A diagram variable is a sequence of capital letters to which, in some cases, a subscript is appended. A diagram variable denotes a specific type of sub-sequence of an AMBIT program in accordance with the informal declarations which precede the ER in which the diagram variable appears. For example, in ER.2.1, "DL" denotes a declarative list and "IL" denotes an imperative list in accordance with the declarations which precede ER.2.1. A diagram literal is a reserved program symbol (see PF.24), a parenthesis, or one of the symbols '∇', '$\mathcal{S}$', '$\mathcal{F}$', or '$\mathcal{R}$'. A diagram literal denotes an instance of itself. Finally, a blank in a diagram denotes a blank or an empty sequence in an AMBIT program.

A scanning rule is "applicable" if there currently exists a sub-sequence of the AMBIT program under execution which is described by the diagram to the

left of '⇒' in the scanning rule. To "apply" a scanning rule, replace the longest sub-sequence of the program which is described by the diagram to the left of '⇒' by a symbol sequence described by the diagram to the right of '⇒'. The process of matching the diagram to the left of '⇒' in the scanning rule with a sub-sequence of the program will assign certain symbol sequences as the "values" of the diagram variables which appear in that diagram; these values are used in forming the replacement for the sub-sequence and are defined for the duration of a given execution rule.

The order in which the execution rules are applied to effect the execution of an AMBIT program is defined by the following algorithm:

1. Let ER.1.1 be called the "current ER" and go to Step 3.
2. Let the ER which follows the current ER be the "current ER".
3. If the scanning rule in the current ER is not applicable to the program under execution, then go to Step 2.
4. If the current ER has one or more conditions and any one of these conditions is not satisfied, then go to Step 2.
5. Apply the scanning diagram to the longest symbol sequence of the program under execution to which it is applicable.
6. If the current ER has one or more "actions", perform these actions in the order in which they are given.
7. Go to Step 1.

2-156

In a well-formed program it will never occur that Step 2 will be attempted when the "current ER" is the last ER (ER.13.6); the PE cease to act and the algorithm above terminates only as the result of the execution of ER.1.2. The only modification of the program performed by the execution rules is the temporary insertion of occasional parenthesis pairs (as by ER.5.1) and of the scanning symbols '∇', '$\mathcal{S}$', '$\mathcal{F}$', and '$\mathcal{A}$'.

The foregoing discussion is adequate for a purely formal interpretation of the ER. However, the ER are rules for the execution of an AMBIT program

which are not only sufficient as a formal definition but which are intended to be convenient and natural for a human program executer. This aspect of the ER will now be considered.

The fundamental action specified by the execution rules as a whole is the advancing of a "control scanner", the symbol '∇', through an AMBIT program. Since the scanner is a symbol which may not legally appear in an AMBIT program, there is no danger of the scanner being confused with the program. The scanner may be thought of as the "focus of attention" of a human program-executer as he reads through and executes an AMBIT program. The first action of the PE is to insert the scanner before the first symbol in the AMBIT program, in accordance with ER.1.1. The last action of the PE is to remove the scanner from after the last symbol of the AMBIT program, in accordance with ER.1.2. Between these first and last actions, there is always exactly one instance of the scanner in the AMBIT program. The execution rules move this scanner from left to right through the program, with the single exception of ER.11.1, which may cause the scanner to jump backward (from right to left).

The scanner is immediately followed by the symbol '$\mathfrak{R}$' when it is moving through a replacement rule. The '$\mathfrak{R}$' indicates that the PE is "reading" the replacement rule; that is, forming an interpretation expressed in conventional English of the action specified by the AMBIT replacement rule. When the scanner reaches the end of the replacement rule, the accumulated interpretation of the replacement rule consists of two English sentences which specify a modification of the current data string. If the modification is currently possible, ER.13.3 causes the '$\mathfrak{R}$' which follows the scanner to be replaced by '$\mathcal{S}$' (indicating that the replacement rule has "succeeded") and causes the PE to perform the specified modification of the data string. Otherwise, ER.13.4 causes '$\mathfrak{R}$' to be replaced by '$\mathcal{F}$' (indicating that the replacement rule has "failed") and causes the PE to leave the data string unchanged.

On occasion during the execution of an AMBIT program, the scanner will immediately follow a sub-sequence of the program which is a complete imperative (or which would be a complete imperative if the scanner were replaced by a

semicolon). On these occasions, the scanner is always followed by the symbol '$\mathcal{S}$' or '$\mathcal{F}$', according as the execution of that complete imperative has "succeeded" or "failed". In the preceding paragraph this convention was established for the replacement rule; but it also holds for implications, disjunctions, conjunctions, negations, primary imperatives, and special procedure calls, all of which, according to PF.11 through PF.20, are imperatives. The "success" or "failure" of these imperatives is compounded in much the same way that the success of complex actions expressed in imperative English is compounded.

ER: Execution Rules for the AMBIT program.

Program. Let PROG be the AMBIT program which is to be executed.

1.1. PROG ⇒ ∇ PROG
condition: No execution rule has previously been applied to the program in this execution of this program.

1.2. PROG ∇,$\mathcal{S}$ ⇒ PROG
action: Expire.

Block. Let DL be a declarative list. Let IL be an imperative list.

2.1. ∇ begin DL IL end ⇒ begin DL ∇ IL end

2.2. begin DL IL ∇ end ⇒ begin DL IL end ∇,$\mathcal{S}$

Attached label. Let ID be an identifier.

3.1. ∇ ID : ⇒ ID : ∇

Imperative Statement. Let IVE be an imperative.

4.1. IVE ∇,$\mathcal{S}$; ⇒ IVE ; ∇

4.2. IVE ∇,$\mathcal{F}$; ⇒ IVE ; ∇
action: Output an error message, such as "Execution-time error; the imperative statement 'IVE ;' is non-executable."

Imperative. Let IMP be an implication.

5.1. ∇ try IMP ⇒ try (∇ IMP)

5.2. try (IMP ∇,$\mathcal{S}$) ⇒ try IMP ∇,$\mathcal{S}$

5.3. try (IMP ∇,$\mathcal{F}$) ⇒ try IMP ∇,$\mathcal{S}$

Implication. Let D_1, D_2, and D_3 be disjunctions.

6.1. ∇ if D_1 then D_2 else $D_3 \Rightarrow$ if ∇ D_1 then D_2 else D_3

6.2. if D_1 $\nabla\mathcal{S}$ then D_2 else $D_3 \Rightarrow$ if D_1 then ∇ D_2 else D_3

6.3. if D_1 $\nabla\mathcal{F}$ then D_2 else $D_3 \Rightarrow$ if D_1 then D_2 else ∇ D_3

6.4. if D_1 then D_2 $\nabla\mathcal{S}$ else $D_3 \Rightarrow$ if D_1 then D_2 else D_3 $\nabla\mathcal{S}$

6.5. if D_1 then D_2 $\nabla\mathcal{F}$ else $D_3 \Rightarrow$ if D_1 then D_2 else D_3 $\nabla\mathcal{F}$

6.6. ∇ if D_1 then $D_2 \Rightarrow$ if ∇ D_1 then D_2

6.7. if D_1 $\nabla\mathcal{S}$ then $D_2 \Rightarrow$ if D_1 then ∇ D_2

6.8. if D_1 $\nabla\mathcal{F}$ then $D_2 \Rightarrow$ if D_1 then D_2 $\nabla\mathcal{S}$

Disjunction. Let C be a conjunction. Let D be a disjunction.

7.1. C $\nabla\mathcal{S}$ or D $\Rightarrow$ C or D $\nabla\mathcal{S}$

7.2. C $\nabla\mathcal{F}$ or D $\Rightarrow$ C or ∇ D

Conjunction. Let N be a negation. Let C be a conjunction.

8.1. N $\nabla\mathcal{S}$ and C $\Rightarrow$ N and ∇ C

8.2. N $\nabla\mathcal{F}$ and C $\Rightarrow$ N and C $\nabla\mathcal{F}$

Negation. Let PI be a primary imperative.

9.1. ∇ not PI $\Rightarrow$ not (∇ PI)

9.2. not (PI $\nabla\mathcal{S}$) $\Rightarrow$ not PI $\nabla\mathcal{F}$

9.3. not (PI $\nabla\mathcal{F}$) $\Rightarrow$ not PI $\nabla\mathcal{S}$

Primary Imperative. Let IVE be an imperative.

10.1. ∇ (IVE) $\Rightarrow$ (∇ IVE)

10.2. (IVE $\nabla\mathcal{S}$) $\Rightarrow$ (IVE) $\nabla\mathcal{S}$

10.3. (IVE $\nabla\mathcal{F}$) $\Rightarrow$ (IVE) $\nabla\mathcal{F}$

Control Imperative. Let ID be an identifier.

11.1. ∇ go to ID $\Rightarrow$ go to ID

action: Insert '∇' immediately after that sub-sequence of the program under execution which is an attached label and which contains a complete instance of ID.

Special Procedure Call. Let SPN be a special procedure name. Let ID be an identifier.

12.1. ∇ SPN (ID) ⇒ SPN (ID) ∇Δ

action: Let PTR be that pointer which is formed by appending 'Δ' to the identifier ID. If there exists a sub-sequence of the current data string which consists of a complete instance of the pointer PTR followed by a blank followed by a complete instance, PHR, of any phrase, then apply the "expand", "pack", or "evaluate" operation (as defined by DN.8 and DN.9) to PHR according as SPN is 'expand', 'pack', or 'evaluate'; otherwise, leave the current data string unchanged.

Replacement Rule. Let SD_1 and SD_2 be string descriptions. Let SEG be a segment.

13.1. ∇ SD_1 → SD_2 ⇒ (∇R SD_1 → SD_2)

action 1: Apply the "space fully" editing operation to the current data string. (See DN.8)

action 2: Prepare to construct an English-language interpretation of the replacement rule; this interpretation will consist of two imperative English sentences. Begin the interpretation with "Find a sub-sequence of the current data string which satisfies the following description:".

13.2. (SD_1 ∇R → SD_2) ⇒ (SD_1 → ∇R SD_2)

action: Continue the interpretation with "Replace the sub-sequence of the current data string just found with a symbol sequence which satisfies the following description:".

13.3. (SD_1 → SD_2 ∇R) ⇒ SD_1 → SD_2 ∇S

condition: The just-completed interpretation specifies a modification of the current data string which can be performed.

action 1: Perform the modification of the current data string specified by the just-completed interpretation.

action 2: Let PTR be any pointer such that two or more separate instances of PTR are contained in the data string. (Multiple instances of a pointer, if they exist, can only have been introduced by the just-executed replacement rule.) For each such PTR, delete every instance of PTR from the data string except that instance (if any) which was inserted by this execution of this replacement rule as a direct result of a separate instance of PTR in the replacement part of this replacement rule.

action 3: Apply the "space legibly" editing operation to the current data string.

13.4 ($SD_1 \rightarrow SD_2$ $\nabla\mathcal{R}$) $\Rightarrow$ $SD_1 \rightarrow SD_2$ $\nabla\mathcal{F}$

condition: The just-completed interpretation specifies a modification of the current data string which cannot be performed.

action: Apply the "space legibly" editing operation to the current data string.

13.5 $\nabla\mathcal{R}$ SEG $\Rightarrow$ SEG $\nabla\mathcal{R}$

action 1: If SEG is the first (leftmost) segment in a string description, then continue the interpretation with "a blank, followed by "; otherwise, leave the interpretation unchanged.

action 2: Act on one of the following:

a. If SEG is not an identifier, then continue the interpretation with "'SEG', followed by ".

b. If SEG is an identifier declared 'literal', then continue the interpretation with "'SEG', followed by".

c. If SEG is an identifier declared 'name', then continue the interpretation with "a symbol sequence which is identical to the current value of the identifier 'SEG', followed by".

d. If SEG is an identifier declared 'DT dummy' (where DT is a data type) and if a complete instance of SEG has not been interpreted previously in this execution of this replacement rule, then continue the interpretation with "a symbol sequence which is an arbitrary DT and which is hereby named 'SEG', followed by".

e. If SEG is an identifier declared 'DT dummy' (where DT is a data type) and if a complete instance of SEG has been interpreted previously in this execution of this replacement rule, then continue the interpretation with "a symbol sequence which is identical to the symbol sequence previously named 'SEG', followed by".

action 3: Act on one of the following:

a. If SEG is an identifier declared 'string dummy', then continue the interpretation with "an empty sequence or a blank according as the string named 'SEG' is or is not an empty sequence."

b. If SEG is not an identifier declared 'string dummy', then continue the interpretation with "a blank, followed by".

action 4: If SEG is the last (rightmost) segment in a string description, then replace the ", followed by" with which the current interpretation ends by "." (Thus completing an English sentence); otherwise, leave the interpretation unchanged.

13.6. ∇ null ⇒ null ∇

action: Act on one of the following:

a. If this instance of 'null' is a citation, then continue the interpretation with "the blank which appears at the right end of the data string.".

b. If this instance of 'null' is a replacement, then continue the interpretation with "a blank.".

Example. Consider the replacement rule on Line 5 of the example program in Sec. (2), namely

$$EQ2\Delta\ Q \rightarrow EQ2\Delta\ p\Delta\ Q$$

In the course of executing this replacement rule, the PE would construct the following English language interpretation of this replacement rule:

2-162

Find a sub-sequence of the current data string which satisfies the following description: a blank, followed by 'EQ2Δ', followed by a blank, followed by a symbol sequence which is an arbitrary phrase and which is hereby named 'Q', followed by a blank. Replace the sub-sequence of the current data string just found with a symbol sequence which satisfies the following description: a blank, followed by 'EQ2Δ', followed by a blank, followed by 'pΔ', followed by a blank, followed by a symbol sequence which is identical to the symbol sequence previously named 'Q', followed by a blank.

An informal interpretation of this replacement rule was given in Sec. (2); the interpretation given here differs primarily in its attention to the use of blanks in the data string.

2-163

4. ON THE DESIGN OF AMBIT.

The fundamental decision on which the design of the AMBIT system was based was the decision to incorporate the conventional identity into a formal programming language. This decision did not carry in itself all of the constraints necessary for the complete definition of a programming language. It was necessary to develop an "algorithmic interpretation" of the identity, to develop a suitable representation for the operand data, and to provide a control environment for the execution of identities within a program. The design of the AMBIT system has been a process of testing various design options, examining the interaction of these options within an experimental programming language, and finally developing general criteria for the acceptance or rejection of given options. The design of AMBIT presented in Sec. (3) incorporates these design decisions, but does not make clear the reasoning behind them. In this section, the design criteria of the current AMBIT system will be discussed informally, and some motivation for the design of the AMBIT system will be provided.

4.1. Three Important Properties of AMBIT.

An AMBIT program is always such that its execution is unique, plausible, and conservative. These properties of AMBIT will be defined in the following paragraphs. They represent important aspects of the discipline which AMBIT imposes on the use of the identity (replacement rule), a discipline which allows the AMBIT programmer to make important assumptions about the execution of a valid AMBIT program. These properties of AMBIT depend, to a large degree, on the restrictions expressed in the DN and the PN; thus a discussion of these properties provides the motivation for the inclusion of those restrictions in the formal definition of AMBIT. The properties discussed in the following paragraphs do not depend solely on the restrictions cited in those paragraphs; for example, they also depend in an important way on the requirement that parentheses match within a string.

The execution of a program is "unique" in the following sense: The execution of a given AMBIT program on a given data string uniquely determines the modified data string produced by the execution of the program (with exceptions to be noted below). The uniqueness of program execution depends on the uniqueness of execution of the individual simple imperatives contained in the program. In this connection, the following assertion is made: The execution of a given simple imperative within an AMBIT program on a given data string uniquely determines (a) the position of the scanner '∇' within the program immediately after execution of the simple imperative and (b) the modified data string produced by the execution of the simple imperative. The DN and PN contribute to the latter assertion as follows: If the simple imperative is a replacement rule, the uniqueness of its execution depends on DN.4 (uniqueness of pointers in the data string), PN.3 (uniqueness of declaration), PN.9 (non-ambiguity of citations), and PN.10 (non-ambiguity of replacements). If the simple imperative is a control imperative, the uniqueness of its execution depends on PN.6 (uniqueness of labelling). If the simple imperative is a special procedure call, the uniqueness of its execution depends on DN.4 (uniqueness of pointers in the data string) and on the uniqueness of the operations defined

by DN.8 (editing operations) and DN.9 (the evaluate operation).

The exceptions to the uniqueness of program execution are two. First, because the data string will be "legibly spaced" at the end of the execution of an AMBIT program, the presence or absence of certain blanks is not determined by the formal definition of AMBIT. This ambiguity can be removed simply by applying the "space fully" operation to the final data string. The ambiguity is superficial, and allows the human program executer to dispense with blanks which, from the standpoint of a human reader, are superfluous. Second, the precision of "real" numbers calculated in accordance with DN.9 (the evaluate operation) is not defined; this ambiguity is accepted in AMBIT for the same reasons it is accepted in ALGOL 60: different program executers have different facilities for the performance of arithmetic.

2-164

2-165

The execution of a program is "plausible" in the following sense: Given any simple imperative in an AMBIT program, there exists some data string such that if that data string were the current data string, the execution of the simple imperative would succeed. (It is <u>not</u> asserted that such a data string will ever become available to the given simple imperative in the course of actual program execution; the program as a whole may contain implausibilities, and the simple imperative may be fore-doomed to perpetual failure.) The plausibility of the replacement rule depends on PN.13 (plausibility of citations); the plausibility of the control imperative depends on PN.7 (existence of labelling). The plausibility of the special procedure calls depends on the obvious plausibility of the operations defined by DN.8 and DN.9.

The execution of a program is "conservative" in the following sense: The execution of a given AMBIT program on a given data string never produces (as a final result) a modification of the data string which is not itself a data string. Thus the execution of an AMBIT program conserves the important properties of the data string: the limited character set, the matching of parentheses, and the uniqueness of pointers. The only structures in an AMBIT program which modify the data string are the replacement rules and the special procedure calls. In this connection, the following assertion is made: The execution of a given re-

placement rule or special procedure call within an AMBIT program never produces (as a final result of the execution) a modification of the data string which is not itself a data string. The conservative execution of a replacement rule depends on PN.11 (conservation of matching parentheses), on PN.12 (conservation of uniqueness of pointers), and on "action 2" of **ER.13.3.** which removes duplicate instances of pointers temporarily inserted into the data string during execution of a replacement rule. The conservative execution of a special procedure call depends on DN.8 (the editing operations), which define operations on the data string designed to avoid the destruction or creation of pointers in the data string.

4.2. Words and Blanks.

The AMBIT system is not designed primarily for the manipulation of individual symbols; rather, the atomic objects manipulated are the objects called "elements". According to DF.5, an element is either a mark (which is a single symbol) or a word (which is not, in general, a single symbol). The choice of the element rather than the character as the atomic object in the data string complicates the definition of AMBIT. It requires the active use of the blank (or some other separating character) to separate two elements which are words and which, if they were immediately adjacent, would merge into a single word.

Conventional algebra almost completely avoids the use of a pure separator such as the blank; it adopts an "operator grammar" such that two words (usually operands) are always separated by a mark (usually an operator). On the other hand, conventional English makes extensive use of the blank. The AMBIT system attempts to extract advantages from both of these systems. A blank is "significant" only when it occurs between two alphanumerics, and is thus available as a pure separator between two words just as in conventional English. On the other hand, a blank between two symbols one of which is an operator is "non-significant", and may be omitted or retained according to the requirements of legibility, just as in conventional algebra.

According to ER.13, the program executer must begin the execution of each replacement rule by applying the "space fully" operation to the data string, and must conclude the execution by applying the "space legibly" operation to the data string. This requirement considerably simplifies the formal definition of the execution of a replacement rule. However, literal compliance with this requirement demands much re-copying of the data string. The human program executer will probably prefer to keep the data string in "legibly spaced" form at all times and to imagine, rather than actually insert, those blanks which would be inserted by the execution of the "space fully" operation.

The 'expand' and 'pack' operations are necessary only if the AMPIT programmer decides to suspend the convention that an element is an atomic structure. The operation 'expand' may then be used to convert a single element into a sequence of elements separated by blanks; and the operation 'pack' may be used to re-assemble the sequence into a single element. These operations would be used, for example, in writing a program for the explicit, digit-by-digit addition of two numbers, or in writing a program to arrange a sequence of identifiers in lexicographical order.

2-166

2-167

4.3. Dummies and Names.

The example program given in Sec. (2) does not illustrate certain important aspects of the replacement rule. Two supplementary examples will be given here. Consider first the replacement rule

$$p\Delta \ \ (A \times A) \rightarrow p\Delta \ \ (A\uparrow 2)$$

and assume that 'A' is declared 'phrase dummy'. If this replacement rule is executed when the data string is (for example)

$$EQ\Delta \ \ ((alpha+p\Delta((m+1) \times (m+1))) = 6.0)$$

then the replacement rule succeeds and the modified data string is

EQΔ ((alpha+pΔ((m+1)⍎2)) = 6.0)

But if the replacement rule is executed when the data string is (for example)

EQΔ ((alpha+pΔ((m+1)×(m+5))) = 6.0)

then the replacement rule fails and the data string is not changed. This example illustrates the use of multiple instances of the same "dummy variable" (an identifier declared 'DT dummy', where DT is a data type) in a single citation. It depends on the fact that the second instance of 'A' in the citation is interpreted by "action 2.e" of ER.13.5 rather than "action 2.d".

As an example of the use of identifiers declared 'name', consider the following imperative:

```
( ( aΔ  a  →  aΔ  a  or  null  →  aΔ  ?  )  and
    aΔ  a  →  aΔ((b+16)×(c⍎2)) and evaluate (a)   )
```

and assume that the identifiers 'a', 'b', and 'c' are all declared 'name'. The execution of this imperative will succeed for any data string whatever. Four examples of the execution of this imperative follow; each is a pair of data strings and represents the data string immediately before and immediately after the successful execution of the imperative.

```
1.  bΔ 4 cΔ(.125+.375) aΔ 1.2
    bΔ 4 cΔ(.125+.375) aΔ 5.0
2.  bΔ 4 cΔ(.125+.375)
    bΔ 4 cΔ(.125+.375) aΔ 5.0
3.  bΔ a cΔ(.125+.375) aΔ 1.2
    bΔ a cΔ(.125+.375) aΔ ((a+16)×(.25))
4.  cΔ(.125 + .375) aΔ 1.2
    cΔ(.125 + .375) aΔ ((?+16)×(.25))
```

These data strings contain only symbol sequences which are relevant to the imperative under consideration; in actual practice, the data shown above would be part of a larger data string containing symbol sequences not affected by the execution of this imperative.

The AMBIT imperative just considered is equivalent to the following ALGOL assignment statement:

$$a := (b+16) \times (c\uparrow 2)$$

where 'a' and 'c' have the ALGOL declaration 'real' and 'b' has the ALGOL declaration 'integer'. The example just given suggests the way in which arithmetic can be performed in AMBIT. More important, it shows that an "assignment statement" similar to that used in ALGOL 60 could be added to an expanded version of the AMBIT language and could be defined in terms of existing AMBIT constructs.

2-169

2-168

The two examples considered in this section show that it is the identifier declared 'name' and not the "dummy variable" which plays the role of the conventional program variable in the AMBIT language.

4.4. The Data Types.

The most arbitrary aspect of the current design of AMBIT is the choice of data types. It is not clear, for example, why there should be a data type "unsigned integer" but no data type "signed integer". It has been necessary to select a small set of data types and incorporate these as a fixed feature of the AMBIT language. It is to be hoped that, in some future version of AMBIT, the programmer will be able to define his own data types and use these in the declaration of identifiers.

The data types may be divided, in a rough way, into two groups: general and mathematical. Typical general data types are "string", "phrase", "element", and "segment". These data types have no special connection with mathematical data, but rather apply to any data in the form of a parenthesized symbol sequence.

Typical mathematical data types are "number", "Boolean", "integer", and "sign". These data types are of special value in describing mathematical data. The distinction is not a perfect one; data types such as "letter" and "digit" are general or mathematical, depending on the programmer's point of view. One of the most important data types is "alphanumeric", which determines the set of data symbols which "run together" to form single elements.

Five of the metavariables defined by the DF are not included in the data types allowed by PF.6 for use in the declaration of identifiers. The omitted metavariables are accounted for as follows: The metavariable "data string" is omitted because it is not intended that the AMBIT programmer shall cause the manipulation of the data string as a whole; in fact, the AMBIT programmer will usually regard the data string as an arbitrarily long symbol sequence the ends of which are not accessible to him. The metavariables "decimal" and "exponent" are omitted because the machine representation of real numbers would make the use of these metavariables as data types in a program a source of inefficiency. The metavariables "parenthesis" and "data symbol" are omitted because their introduction requires additional measures to assure the conservation of the data string, and these complications are not worth the small advantage in data-type vocabulary gained.

4.5. Extended Program Logic.

In ALGOL 60, the data test (the Boolean expression) and the data modification (the assignment statement) are distinct features of the language; and program logic is limited to the use of implication (the "if...then..." structure and the "if...then...else..." variant of this structure). In most cases, data tests do not modify data and data modifications do not test data. Early versions of the AMBIT programming language followed ALGOL in this separation of data test from data modification. But example programs written in those versions of AMBIT were redundant and unclear. It was observed that while the separation of data test and data modification was appropriate for the performance of arith-

metic, it was not appropriate for symbol manipulation. Therefore a program logic which abandoned this separation was adopted; and it is this "extended program logic" which is incorporated in the present AMBIT language.

The replacement rule is the combined data test and data modification facility of the AMBIT programming language. The replacement rule is a data test because the "success" or "failure" of its execution, which depends on the contents of the data string, may be used to modify the flow of control in the program. The Boolean (two-valued) character of the replacement rule suggested that a full complement of logical operators should be introduced into the program logic of AMBIT. However, logical operators were introduced into AMBIT only when their usefulness had been demonstrated by their use in example programs. On this basis, negation, conjunction, disjunction, implication, and the 'try' operator were admitted to AMBIT program logic and equivalence was excluded. The replacement rules thus connected by logical operators specify modifications of the data string, and a convention for the use of these operators is required to determine the order in which the replacement rules are executed. The convention chosen is a familiar one. It not only has some precedent in the interpretation of imperative English sentences, but conforms to the extensively documented technique for the evaluation of Boolean expressions in ALGOL 60 [6]. This convention is expressed in ER.6 through ER.9.

The AMBIT language does contain a pure data test, although its presence in the language is not obvious. A replacement rule which has a replacement which is identical to its citation is a pure data test. If the sub-sequence described by the citation of the replacement rule is present in the data string, the replacement rule succeeds and the sub-sequence is replaced by an identical sub-sequence. If the sub-sequence described by the citation is not present in the data string, the replacement rule fails and no modification of the data string is performed. In both cases, the net effect is to leave the data string unchanged and to cause a branching of the flow of program control based on the contents of the data string. Experimental programming in AMBIT has shown that the requirement for a pure data test is surprisingly rare.

4.6. The Computer Implementation of AMBIT.

The AMBIT system has not been implemented for an automatic computer. However, the design of the AMBIT system was carried out with careful attention to the characteristics of existing computer hardware, and an efficient implementation of the system has been planned. A detailed discussion of this plan will not be given here; instead, the representation of the data string in computer memory will be described. The computer representation of the data string, together with an algorithm based on PN.9 (non-ambiguity of the citation) constitute the basis for an efficient implementation of AMBIT.

The data string is stored in computer memory in the form of a "symmetric threaded list". The list is composed of "cells", each of which represents a single complete instance of a segment or a pointer in the data string. In the memory of a typical computer (for example, the IBM 7090), each cell occupies two adjacent memory locations. A cell which appears at locations i and i+1 of the computer memory consists of the following fields:

1. The Contents Field, C(i). This field contains a binary encodement of one segment of the data string, or else an address through which that encodement can be found.
2. The Left Link Field, L(i). This field contains the address of another cell, namely that cell which contains the left neighbor of C(i).
3. The Right Link Field, R(i). This field contains the address of another cell, namely that cell which contains the right neighbor of C(i).
4. The Parenthesis Link Field, P(i). This field is defined if and only if C(i) is a parenthesis. It contains the address of the cell which contains the matching parenthesis to C(i).
5. The Property Field, Pr(i). This field contains bits which indicate the syntactic type (or "data type") of C(i).

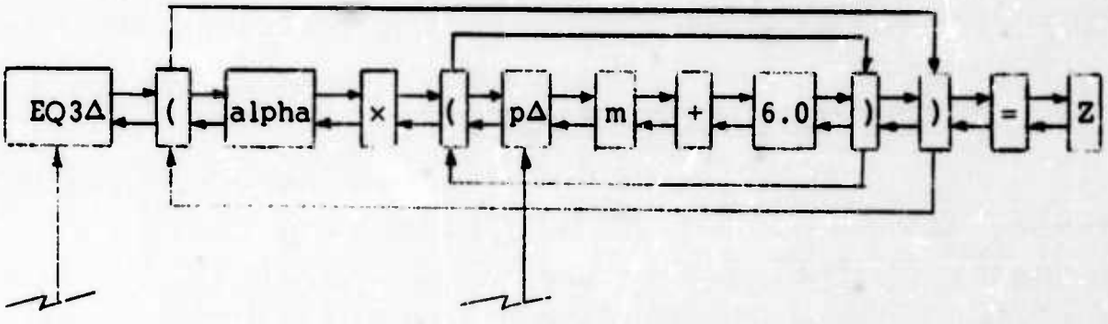
In general, the AMBIT program executer has no information about the location of a particular cell in the computer representation of the data string. The exception is a cell which contains a pointer; such a cell is assigned a permanent and recorded location by the AMBIT compiler, and the program executer has direct access to the cell.

Consider the following data string

EQ3Δ (alpha × (pΔ m+6.0)) = Z

the computer memory representation of this data string as a symmetric threaded list may be diagrammed as follows:

2-173



In this diagram the Contents Field is shown literally (but not in a binary encodement); the Left Link, Right Link, and Parenthesis Link Fields are shown by arrows; the Property Field is omitted; and the direct accessibility of the pointers is shown by arrows entering from outside the diagram.

The citation of a replacement rule is a description of a sub-sequence of the data string which is to be located by the program executer. If it were necessary for the executer to search the entire data string, segment by segment, for the sub-sequence specified, then an efficient implementation of AMBIT would be impossible. However, in inspecting the data string the program executer not only makes use of Right and Left Links to scan through the data string, but also uses directly accessible pointers to enter the data string and uses parenthesis links to skip across arbitrary sub-strings of the data string. This

facility permits an efficient implementation of the AMBIT system.

The type of list structure just defined is well known. It most closely resembles the "threaded list" of Perlis and Thornton [7], but is "symmetric" in the sense that it may be scanned with equal facility in either direction. It resembles the "symmetric list" of Weizenbaum [8], but it does not allow the use of a single instance of a sub-list in two separate lists. In practice, a pair of cells containing a pair of matched parentheses might be merged into a single cell, and pointers might not be linked directly into the data string; but these are implementation tricks which, while they would allow a somewhat more compact memory-representation of the data string, do not affect the definition of the AMBIT language. The design of an AMBIT compiler is certainly not obvious from the brief discussion of the data representation given here; nevertheless, the formal definition of AMBIT and the data representation given here largely determine such a design.

5. ON THE EXTENSION OF THE AMBIT SYSTEM.

The AMBIT system defined in this paper is a relatively simple programming system. To some extent, this simplicity has been achieved by the exclusion of facilities which are clearly useful and necessary in a general system for symbol manipulation. These facilities were excluded from the AMBIT system defined in this paper in order to permit a complete and formal definition of the system; the definition of a complete system for symbol manipulation would have been either excessively long or considerably less precise. Furthermore, the definition of a complete system would include definitions of facilities which are not closely related to the incorporation of an algorithmic interpretation of the identity in a programming language, and which are thus beyond the ambit of this paper.

In this section, the characteristics of a complete system for symbol manipulation based on extensions of the AMBIT system will be discussed. The extensions considered are of two kinds: remedial and developmental. The remedial extensionsof AMBIT are facilities which are well known and which play a useful role in familiar programming languages; important remedial extensions are

as follows:

1. The introduction of "block structure" to control the scope of the declaration of identifiers within a program, as in ALGOL 60. (Although block structure exists in the current AMBIT, it is not fully exploited.)

2. The introduction of a convention for the definition of self-contained procedures within a program, as in ALGOL 60.

3. The introduction of quotation conventions to allow the use of all available symbols in the data string (in particular, the reserved program symbols, 'begin', ';', etc.).

4. The introduction of a specialized facility for programming purely arithmetic operations, based on the "assignment statement" of ALGOL 60 and defined in terms of existing AMBIT facilities (see Sec. 4.3).

5. The introduction of subscripted identifiers (declared 'name') to permit the convenient processing of arrays.

6. The introduction of a more general control imperative.

7. The introduction of input-output facilities for the transmission of all or part of the data string to and from data locations external to the current data string. These facilities should include operations to control the two-dimensional format and spacing of data in its input or output form (external to the system) and conversion operations to permit the input and output of data which, in its representation external to the system, is not a legal data string.

The developmental extensions of AMBIT are facilities which are suggested by the refinement or generalization of the current AMBIT system rather than the inspection of other programming languages. Possible developmental extensions are as follows:

1. The introduction of notations for linking non-adjacent portions of the data string which are more general than the parenthesis pair, and of notations

for entry into the data string which are more general than the pointer.

2. The introduction of a facility by which the programmer may define ad hoc data types (by submitting the necessary metalinguistic formulae with his program) and then use these data types in declaring dummy variables (see Sec. 4.4).

3. The introduction of a notation similar to the replacement rule to specify the transmission of data between the data string and data locations external to the data string, thus utilizing an identity-like notation for input-output as well as internal operations on data.

4. The introduction of certain refinements in the notation for the replacement rule, such as the use of the ellipsis symbol '...' as a special form of string dummy variable.

It is believed that the extension of AMBIT can be achieved in a way which will emphasize rather than obscure the central importance of the algorithmic interpretation of the identity in the design of the system.

2-176

2-177

REFERENCES

1. Floyd, R. W. A descriptive language for symbol manipulation. J.ACM 8 (October 1961), 579-584.

2. COMIT Programmer's Reference Manual. (Second edition) MIT Press, Cambridge, Mass., 1961.

3. Farber, D. J., Griswold, R. E., and Polonsky, I. P. SNOBOL, a string manipulation language. J.ACM 11 (January 1964), 21-30.

4. Perlis, A. J., and Iturriaga, Renato. An extension to ALGOL for manipulating formulae. Comm ACM 7 (February 1964), 127-130.

5. Naur, Peter (Ed.). Revised report on the algorithmic language ALGOL 60. Comm ACM 6 (January 1963), 1-17.

6. Huskey, Harry D. and Wattenburg, W. H. Compiling techniques for Boolean expressions and conditional statements in ALGOL 60. Comm ACM 4 (January 1961), 70-75.

7. Perlis, A. J. and Thornton, Charles. Symbol manipulation by threaded lists. Comm ACM 4 (April 1960), 195-204.

8. Weizenbaum, J. Symmetric list processor. Comm ACM 6 (September 1963), 524-544.

EXAMPLES OF SYMBOL MANIPULATION IN THE AMBIT PROGRAMMING LANGUAGE

by

Carlos Christensen

Research Paper
CA-6504-0111

February 26, 1965

To be presented at the ACM 20th National Conference, Aug. 1965

The research reported in this paper was sponsored in part by the Air Force Cambridge Research Laboratories, Office of Aerospace Research, under Contract AF19(628)-419, and by the Rome Air Development Center, under Contract AF30(602)-3342.

ABSTRACT

This paper presents several examples of programs written in a programming language system called AMBIT (Algebraic Manipulation By Identity Translation). AMBIT is related to list processing and algebraic symbol manipulation as FORTRAN and ALGOL are related to numeric calculation. AMBIT is based on a form of conventional notation, the "identity", just as FORTRAN and ALGOL are based on a form of conventional notation, the "formula". The examples of AMBIT programs given in this paper indicate the scope of application of AMBIT and provide an introduction to AMBIT programming. The paper contains a description of AMBIT which is sufficient for an understanding of the examples, but does not contain a complete and formal definition of AMBIT.

2-183

CONTENTS

1. INTRODUCTION.

The AMBIT programming language has been applied to problems in two areas. In the area of algebraic symbol manipulation, programs have been written in AMBIT for elementary arithmetic, elementary algebra, formal differentiation, set operations, and propositional calculus. In the area of list processing, a variety of programs have been written for syntactic analysis, including both "top-down" and "bottom-up" analysis. This programming effort was concurrent with the design of AMBIT and provided practical guidance for that design. It is believed that AMBIT is also appropriate for the programming of heuristic processes, such as formal integration, simplification of algebraic equations, and theorem proving.

A complete and formal definition of a version of AMBIT exists [1]. This paper is intended to exhibit the important features of the AMBIT programming language without entering into a complete and formal definition of the language. The paper contains a brief informal definition of AMBIT (Sec. 2) and several examples of programs written in AMBIT (Sec. 3). The informal definition of AMBIT depends on the example programs for the illustration and clarification of difficult points. The reader may find it useful to defer a detailed examination of Secs. 2.4 and 2.5 of the definition of AMBIT until he has read through the example programs of Sec. 3.

AMBIT has not yet been implemented for an automatic computer; however, it has been shown that AMBIT programs can be compiled into efficient computer programs. AMBIT is related to certain earlier programming languages [2,3,4,5] which make use of an identity-like notation rather than subroutine calls or recursive functions to specify symbol manipulation.

2. AN INFORMAL DEFINITION OF AMBIT.

The central feature of the AMBIT system is the "replacement rule". A replacement rule is a command which specifies the modification of a symbol sequence. It resembles the "identity" of conventional mathematical notation, but its interpretation is extended and rendered explicit by means of a formal definition. The remainder of the design of the AMBIT system

follows from the requirements of the replacement rule. That is, the efficient use of the replacement rule requires that the symbol sequence on which it operates have special syntactic properties; and the control of a collection of replacement rules requires that a program logic exist in which the replacement rules can be embedded.

The AMBIT system consists of three parts: a programming language, a data language, and a program executer. The programming language is a set of symbol sequences each of which is an AMBIT program; the data language is a set of symbol sequences each of which is an AMBIT "data string"; and the program executer is an agent which operates on a data string under the guidance of a program.

An AMBIT program describes operations on a single data object, the data string. Execution of a program proceeds as follows: The program executer inputs an AMBIT program and a data string; then the program executer performs successive modifications on the data string by reading through the program and obeying the replacement rules therein; and finally the program executer outputs the given AMBIT program and the modified data string.

2.1 Data Types.

The AMBIT programming language is a language for the description of operations on symbol sequences. It is appropriate that a discussion of AMBIT begin with the definition of certain formally defined names for the symbol sequences which are to be manipulated. Each word in this formal vocabulary for the description of symbol sequences is called a "data type". Some of the data types used in AMBIT are as follows:

letter -- any one of the 52 upper and lower case letters of the alphabet.

digit -- any one of the ten decimal digits.

Boolean -- either of the special symbols 'true' and 'false'.

alphanumeric -- a letter, a digit, a Boolean, '.', '?', or 'Δ'.

word -- a sequence of one or more alphanumerics

number -- a sequence of symbols which contains one or more digits and, optionally, a decimal point '.'.

identifier -- a letter followed by a sequence of any number (possibly zero) of letters and digits.

pointer -- an identifier followed by 'Δ' followed by any number (possibly zero) of digits.

sign -- '+' or '-'.

arithmetic -- a sign, '×', '/', or '↑'.

relational -- '<', '≤', '=', '≥', '>', or '≠'.

logical -- '≡', '⊃', '∨', '∧', or '¬'.

mark -- an aritnmetic, a relational, or a logical.

character -- an alphanumeric or a mark.

element -- a word or a mark.

phrase -- an element or '(' followed by a string followed by ')'.

segment -- an element, '(', or ')'.

string -- a sequence of any number (possibly zero) of phrases and blanks.

blank -- an unmarked space within a printed line or the break in a symbol sequence which occurs in passing from one line of printing to the next.

The character set implied by the definitions above consists of 85 non-blank symbols and the blank. In a computer implementation of AMBIT, the size of the character set might be reduced by replacing some of the symbols with key identifiers and eliminating lower case letters. On the other hand, for applications other than those discussed in this paper, the character set and also the vocabulary of data types might be extended; in fact, the complete definition of AMBIT [1] makes use of additional symbols and additional

data types. The selection of the character set and the vocabulary of data types is somewhat arbitrary, and can be modified without a fundamental change in AMBIT.

Note that the definition of the data type "string" given above specifies indirectly that a string shall not contain parentheses unless those parentheses occur in "matched" pairs. This assertion follows from the fact that '(' and ')' can only be introduced into a string in matched pairs as the first and last symbols, respectively, in a phrase. Thus the term "string" is used in a special sense in this paper and in AMBIT, while the term "symbol sequence" is used informally to designate any sequence of any symbols.

Note further that the data type "string" can designate an empty symbol sequence. Thus it is possible to imagine the presence of a string (in the form of an empty sequence) between any two adjacent symbols in a symbol sequence. The data type "string" is the only data type which can designate an empty sequence.

Some additional terminology will be necessary. A "separate instance of an identifier" is an instance of an identifier which is not embedded in a longer sequence of alphanumerics. A "separate instance of a pointer" is defined in the same way. For example, the symbol sequence '1+a bcΔ=3' contains two instances of a pointer, 'bcΔ' and 'cΔ', but only one separate instance of a pointer, 'bcΔ'; similarly, it contains four instances of an identifier, 'a', 'b', 'bc', and 'c', but only one separate instance of an identifier, 'a'.

2.2 AMBIT Equivalence.

In order to describe the manipulation of symbol sequences, it is necessary to have some way of comparing symbol sequences to determine whether, in some sense, they are equivalent. A simple equivalence relation between symbol sequences would be one which required them to consist of the same symbols in the same order. The equivalence relation

used in the AMBIT system, called "AMBIT equivalence" is somewhat more complicated; specifically, it is designed to permit certain uses of the blank to increase the legibility of a symbol sequence without affecting the formal content of the symbol sequence.

Two instances of symbol sequences are "AMBIT equivalent" if and only if it is possible to transform the first symbol sequence into the second by the application of some combination of the following rules:

1. Delete a blank which separates two symbols in the symbol sequence unless both symbols are alphanumerics.

2. Insert a blank between two immediately adjacent symbols in the symbol sequence unless both symbols are alphanumerics.

3. Replace a blank by a larger or smaller blank.

It follows from the definition just given that the relation of AMBIT equivalence is reflexive, symmetric and transitive.

Some examples of AMBIT equivalence follow. Let the following symbol sequence be called "sequence A":

sin u = (2.83xv) + w

Sequence A is AMBIT equivalent to

sin u = (2.83 x v)+w

because two applications of Rule 1 (deletion of blanks), four applications of Rule 2 (insertion of blanks), and one application of Rule 3 (expansion of blanks) transform sequence A into the symbol sequence just given. Sequence A is not AMBIT equivalent to

sinu = (2.83xv) + w

because none of the rules permit the deletion of a blank between the two alphanumeric symbols 'n' and 'u' in sequence A. Similarly, sequence A is not AMBIT equivalent to

s in u = (2.8 3xv) + w

since the rules do not permit the insertion of a blank between either of the symbol pairs 'si' or '83'.

2.3 The Data String.

The data string is that symbol sequence which is operated upon by the program executer during the execution of a given AMBIT program. A symbol sequence is a data string only if it satisfies the following restrictions:

R1. The symbol sequence must be a string and must begin and end with a blank.

R2. The symbol sequence must not contain two separate instances of the same pointer.

R3. The symbol sequence must not contain an instance of a pointer which is neither a separate instance of a pointer nor a subsequence of a separate instance of a pointer.

These restrictions have important implications. According to R2, if a program executer has a pointer in hand, then that pointer either uniquely determines a certain position in the data string or else is totally absent from the data string. According to R1, if the program executer already knows the position of a particular parenthesis in the data string, then the program executer can uniquely determine the position of the matching parenthesis in the data string; this follows from the fact that, according to Sec. (2.1), parentheses exist only in matching pairs in a string. The remaining restriction, R3, is less fundamental; it prevents a pointer from being embedded in a sequence of alphanumerics which is not a pointer, a circumstance which would be notationally unattractive.

2.4 The AMBIT Program.

The general structure of the AMBIT program is derived from ALGOL 60. An AMBIT program consists of 'begin' followed by a list of "declarative statements" followed by a list of "imperative statements" followed by 'end'. Each imperative statement may be preceded by any number of "attached labels" of the form 'ID:' where ID represents any identifier. The declarative statements supply information about the identifiers used in the imperative statements, and are consulted by the program executer whenever this information is required. The execution of the program consists of the sequential execution of the list of imperative statements except where the execution of a "control imperative" (see below) causes a departure from sequential execution.

A declarative statement consists of a "declarator" followed by a list of identifiers followed by a semicolon. The declarator may be 'DT dummy', 'DT name', 'multiple DT name', or 'literal', where DT is any one of the data types defined in Sec.(2.1) except 'blank'. If a declarative statement begins with the declarator D, then each of the identifiers in the statement is said to be "declared D".

An imperative statement consists of a "compound imperative" followed by a semicolon. A compound imperative is composed of "simple imperatives". A simple imperative specifies an operation on the data string or a transfer of control within the program. It may be the case that, in the course of program execution, the action specified by a simple imperative cannot be performed. The program executer not only attempts to perform the action specified by a simple imperative, but also assigns an "execution value", namely 'true' or 'false', to the simple imperative according as the attempt is successful or not.

A compound imperative is one of the following forms: 'SI', '(CI_1)', 'not CI_1', 'CI_1 and CI_2', 'CI_1 or CI_2', 'if CI_1 then CI_2', or 'if CI_1 then CI_2 else CI_3'; where SI is any simple imperative and CI_1, CI_2, and CI_3 are any compound imperatives. These forms are listed in order of decreasing precedence; thus, for example, 'CI_1 and not CI_2 or CI_3' means

'(CI_1 and (not CI_2)) or CI_3'. A compound imperative assumes an execution value based on the execution values of its constituent simple imperatives in accordance with the usual interpretation of the logical words 'not', 'and', etc.

The simple imperatives in an imperative statement are evaluated in left-to-right order. However, if a simple imperative is encountered which (on the basis of the execution values of the simple imperatives already executed) cannot affect the execution value of the imperative statement as a whole, then the simple imperative is skipped over rather than executed. Thus the execution value of a particular simple imperative affects the flow of control through subsequent simple imperatives. An execution value of 'false' is not acceptable for an imperative statement as a whole, and causes the program executer to signal an "execution-time program error" before proceeding to the next imperative statement.

2.5 Simple Imperatives.

Three kinds of simple imperatives will be considered here, the "control imperative", the "replacement rule", and the "existence Boolean". The control imperative is of the form 'go to ID', where ID is any identifier; its execution causes transfer of control to that point in the program which is preceded by the attached label 'ID:'. In a legal AMBIT program there must, in fact, be exactly one such attached label; and it follows that the execution value of a control imperative is always 'true'.

The replacement rule consists of a "citation" followed by the special symbol '→' followed by a "replacement". Both the citation and the replacement must be symbol sequences which satisfy the restrictions placed on the data string (see Sec. 2.3). A replacement rule is executed by finding, if possible, a sub-sequence of the current data string which is AMBIT equivalent to the symbol sequence designated by the citation and replacing that sub-sequence of the data string by a symbol sequence which is AMBIT-equivalent to the symbol sequence designated by the replacement. If this action is possible, then the action is performed and the replacement rule

has the execution value 'true'. If this action is not possible, the current data string is left unchanged and the replacement rule has the execution value 'false'. The execution of the replacement rule is subject to the following rules of interpretation:

R1. A separate instance of an identifier ID which is declared 'DT dummy' designates an arbitrary symbol sequence of data type DT. Two or more separate instances of ID in a single replacement rule designate two or more instances of the same arbitrary symbol sequence of data type DT.

R2. A separate instance of an identifier ID which is declared 'DT name' designates a symbol sequence which is the "current value" of ID. The current value of ID is a symbol sequence S of data type DT which is such that the current data string contains a sub-sequence which is AMBIT equivalent to ' IDΔ S ' (note the initial and final blanks); or, if no such S exists, then the current value of ID is '?'. For example, let 'a', 'b', 'c', and 'd' be identifiers each of which is declared 'phrase name', and let the current data string be

aΔ(ALPHA+bΔ 3.5 cΔ)

Then the current values of the four identifiers are '(ALPHA+bΔ 3.5 cΔ)', '3.5', '?', respectively.

R3. A separate instance of an identifier ID which is declared 'literal' is interpreted literally.

R4. A "separate instance of an ellipsis" is a sequence of three periods which is neither preceded by nor followed by a period. In a legal replacement rule, the number of separate instances in the citation and in the replacement must be the same. Let EC_j be the j-th separate instance of an ellipsis in the citation (counting from the left end of the citation) and let ER_j be the j-th separate instance of an ellipsis in the replacement (counting from the left end of the replacement). Then EC_j and ER_j are interpreted as if they were the only two instances of an identifier declared 'string dummy' (see R1, above).

R5. Let IDΔ be a separate instance of a pointer such that ID is declared 'DT name' or is not declared. Then IDΔ is interpreted literally.

R6. Let IDΔ be a pointer such that ID is declared 'multiple DT name' where DT is a data type. Such a pointer is called a "multiple pointer". Let m be the largest integer such that a separate instance of IDΔm exists in the current data string; or, if no such integer exists, let m = 0. Let n be the number of separate instances of IDΔ in the citation of the replacement rule. Then the j-th separate instance of IDΔ in the citation (counting from the left end of the citation) designates IDΔi, where i is the integer (m-n+j). Similarly, the j-th separate instance of IDΔ in the replacement (counting from the left end of the replacement) designates IDΔi, where i is the integer (m-n+j). For example, let 'p' be declared 'multiple segment pointer' and let the current data string be such that m=4. Then the replacement rule

pΔ pΔ → pΔ pΔ pΔ

will be interpreted as

pΔ3 pΔ4 → pΔ3 pΔ4 pΔ5

R7. All symbols which are not included in a separate instance of an identifier, an ellipsis, or a pointer are interpreted literally. Blanks are always interpreted literally.

The execution of a replacement rule may tend to introduce into the data string two or more separate instances of the same pointer, P. In this case, any separate instances of P which were not introduced under R5 or R6 by the current execution of the replacement rule are deleted from the data string.

In order to be a legal AMBIT replacement rule, a replacement rule must be unambiguous, conservative, and plausible. A replacement rule is "unambiguous" if there exists no data string such that the execution of the replacement rule on that data string could produce either of two distinct modified data strings. A replacement rule is "conservative" if there exists

no data string such that the execution of the replacement rule on that data string produces a symbol sequence which is not a data string. A replacement rule is plausible if there exists at least one data string such that the execution of the replacement rule on that data string has execution value 'true'. These restrictions are independent of the particular data string on which an AMBIT program is actually executed; they are expressed as restrictions on the syntax of the replacement rule in the full definition of AMBIT [1]. All of the replacement rules in the example programs of this paper satisfy these restrictions.

The existence Boolean consists of the special symbol '∃' followed by a citation. The execution of an existence Boolean '∃ C', where C is a citation, is identical to the execution of the replacement rule 'C → C'. Thus the existence Boolean does not modify the data string but may be used as a pure data test to control the flow of the program. An existence Boolean is legal only if the corresponding replacement rule is legal.

2-197

2-196

3. EXAMPLE PROGRAMS.

This portion of the paper is a collection of examples of AMBIT programs. The examples include programs for algebraic symbol manipulation, list processing, and syntactic analysis. Each example program is discussed according to the following outline:

Tne Problem -- The symbol manipulation process performed by the AMBIT program is informally defined.

The Program -- The AMBIT program is given. The lines of the program are numbered sequentially along the left margin; these line numbers are an expository device, and are not a part of the AMBIT program itself.

Exposition -- An interpretation of the program is given.

Example Problems -- In each example problem, a pair of data strings is given. The first data string of the pair is an initial data string (input) for the program, and the second is the corresponding final data string (output).

Trace of an Example Problem -- For some of the programs a trace of the execution of the program on an example data string is given. Each line consists of an "execution history" and a copy of the current data string. For example, the execution history '4f, 16t' means "the simple imperative on Line 4 of the example program was executed and had execution value 'false' (i.e., Line 4 failed), and then the simple imperative on Line 16 was executed and had execution value 'true' (i.e., Line 16 succeeded). This gave the following as the current data string:". An execution history '10×(4f, 16t)' means that the sequence enclosed in parentheses was performed ten times.

3.1 Example 1: Clear Fractions.

The Problem. Given an equation in suitable notation, transform the equation into an algebraically equivalent equation which contains no division operators or which contains only division operators which cannot be cleared by general methods. Both the given equation and the result equation will be fully parenthesized.

The Program.

```
1.          begin string dummy S;   phrase dummy A, B, C;
2.                sign dummy sign;   segment dummy seg;
3.  ENTER:  Given∆(S) → Given∆(S p∆) ;
4.  LOOP:   if ∃ / p∆
5.          then (A /p∆ B)=C      → A p∆ =(B×C)
6.          or   (A /p∆ B)sign C  → (A sign(B×C)) /p∆ B
7.          or   (A /p∆ B)×C      → (A×C) /p∆ B
8.          or   (A /p∆ B)/C      → A /p∆ (B×C)
9.          or   (A /p∆ B)↑C      → (A↑C) /p∆ (B↑C)
10.         or   A=(B /p∆ C)      → (A×C)=B p∆
11.         or   A sign(B /p∆ C)  → ((A×C)sign B) /p∆ C
12.         or   A×(B /p∆ C)      → (A×B) /p∆ C
13.         or   A/(B /p∆ C)      → (A×C)/B p∆
14.         or   sign(B /p∆ C)    → (sign B) /p∆ C
15.         or   seg p∆ → p∆ seg
16.         else seg p∆ → p∆ seg ;
17.         Given∆(p∆ S) → Result∆(S) or
18.         go to LOOP ; end
```

Exposition. The program incorporates ten identities for the clearing of fractions (Lines 5-14). Each of these identities either eliminates a division operator from the equation or moves a division operator one level outward in the parenthesis structure of the equation. The application of the identities is controlled by a scanning pointer, 'p∆', which moves from right to left through the equation. The following is a line-by-line interpretation of the program:

2-198

2-199

Lines 1 - 2. Note that the identifier 'S' designates an arbitrary string; that 'A', 'B', and 'C' each designate an arbitrary phrase; that 'sign' designates an arbitrary sign; and that 'seg' designates an arbitrary segment.

Line 3. Insert 'pΔ' at the right end of the given equation.

Line 4. If 'pΔ' is preceded by '/', then proceed to Line 5; otherwise, skip to Line 16.

Lines 5 - 14. Read through the ten clear-fraction identities in the order in which they are given. If an identity which applies is found, then apply the identity, position 'pΔ' so that it is to the right of any '/' not yet inspected, and skip to Line 17; otherwise, proceed to Line 15.

Line 15. The '/' which precedes 'pΔ' appears in a context from which it cannot be cleared; abandon it, move the 'pΔ' one segment to the left, and skip to Line 17.

Line 16. The pointer 'pΔ' was not preceded by '/'; move 'pΔ' one segment to the left and proceed to Line 17.

Lines 17 - 18. If 'pΔ' has reached the left end of the equation, then delete 'pΔ', identify the result, and exit. Otherwise, go to Line 4.

Example Problems. In example problems E1 and E2, no division operators appear in the result equation; but in E3 a division operator appears in the result as the principal operator of the argument of 'sin' because no general rule exists for the clearing of such a division operator.

E1. GivenΔ(((a/3)+b)=(beta-10))
ResultΔ((a+(3×b))=(3×(beta-10)))

E2. GivenΔ(((2.8×m1)-((a/(b+2))×m2))=((sin(s+t))/(cos(t))))
ResultΔ(((((2.8×m1)×(b+2))-(a×m2))×(cos(t)))=((b+2)×(sin(s+t))))

E3. GivenΔ(((2/3)×(sin((1/m)-(1/n))))=Q)
ResultΔ((2×(sin(((1×n)-(m×1))/(m+n))))=(3×Q))

2-200

2-201

Trace of Example Problem 1.

1.	(input)	GivenΔ(((a/3)+b)=(beta-10))
2.	3t	GivenΔ(((a/3)+b)=(beta-10) pΔ)
3.	4f,16t,17f,18t	GivenΔ(((a/3)+b)=(beta-10 pΔ))
4.	4f,16t,17f,18t	GivenΔ(((a/3)+b)=(beta- pΔ 10))
5.	4f,16t,17f,18t	GivenΔ(((a/3)+b)=(beta pΔ -10))
6.	4f,16t,17f,18t	GivenΔ(((a/3)+b)=(pΔ beta-10))
7.	7×(4f,16t,17f,18t)	GivenΔ(((a /pΔ 3)+b)=(beta-10))
8.	4t,5f,6t,17f,18t	GivenΔ(((a+(3×b)) /pΔ 3)=(beta-10))
9.	4t,5t,17f,18t	GivenΔ((a+(3×b)) pΔ =(3×(beta-10)))
10.	8×(4f,16t,17f,18t)	GivenΔ((pΔ a+(3×b))=(3×(beta-10)))
11.	4f,16t	GivenΔ(pΔ (a+(3×b))=(3×(beta-10)))
12.	17t (output)	GivenΔ((a+(3×b))=(3×(beta-10)))

3.2. Example 2; Differentiate.

The Problem. Given an input data string of the form 'GivenΔ(d(s)/d(var))', where s is a fully parenthesized algebraic equation and var is a variable, carry out the indicated differentiation and return the result as the output data string.

2-201

The Program.

```
1.          begin string dummy s;   phrase dummy u, v;
2.                number dummy const;   word dummy var;
3.                multiple phrase name p;   phrase name x;
4.                literal d;
5.  ENTER:  Given∆(d(s)/d(var)) → Given∆ p∆(s) x∆ var ;
6.  DIF:    p∆(u = v)     →   (p∆ u = p∆ v) or
7.          p∆(u sign v)  →   (p∆ u sign p∆ v) or
8.          p∆(u × v)     →   ((p∆ u × v)+(u × p∆ v)) or
9.          p∆(u / v)     →   (((p∆ u × v)-(u × p∆ v))/(v↑2)) or
10.         [etc.]
11.         p∆(s)       →  (d(s)/d(x)) or
12.         p∆ const    →  0 or
13.         p∆ x        →  1 or
14.         p∆ var      →  (d(var)/d(x)) ;
15.         if ∃ p∆ then go to DIF;
16.         Given∆(s) x∆ var → Result∆(s) ;   end
```

2-202

Exposition. A number of programs for formal differentiation have appeared in the literature; it has been a popular example of recursive programming. Differentiation is included here because the recursion which is appropriate to the problem is performed by means of a "multiple pointer", namely 'p∆', rather than by recursive execution of the program. A line-by-line interpretation of the program follows; throughout this interpretation, the letter 'm' designates the largest integer such that a separate instance of 'p∆m' exists in the current data string (See R6 of Sec. 2.5).

Lines 1 - 4. (The necessary declarations of identifiers are noted.)

Line 5. Select the equation to be differentiated by 'p∆1'. Save the variable of differentiation as the "current value" of the identifier 'x' (See R2 of Sec. 2.5).

Lines 6 - 9. The pointer 'pΔm' selects an expression which has a binary operator. Differentiate the expression and select its operands by by 'pΔm' and 'pΔr', where r=m+1.

Line 10. (The 'etc.' represents additional identities for the differentiation of a unary sign, an exponentiation operator, trigonometric functions, etc.)

Line 11. The pointer 'pΔm' selects an expression for which no differentiation rule has been supplied. Indicate the required differentiation and delete 'pΔm' from the data string.

Lines 12 - 14. The pointer 'pΔm' selects an elementary operand which is a constant, the variable of differentiation, or some other variable. Differentiate the elementary operand and delete 'pΔm' from the data string.

Lines 15 - 16. If the pointer 'pΔm' (for some m>0) exists in the data string, then go back to Line 6; otherwise, discard the variable of differentiation, identify the result, and exit.

Trace of an Example Problem.

1.	(input)		GivenΔ(d(y=((t+4)×t))/d(t))
2.	5t		GivenΔ pΔ1 (y=((t+4)×t)) xΔ t
3.	6t		GivenΔ(pΔ1 y= pΔ2 ((t+4)×t)) xΔ t
4.	15t,6f,7f,8t		GivenΔ(pΔ1 y=((pΔ2 (t+4)×t)+((t+4)× pΔ3 t))) xΔ t
5.	15t,6f...12f,13t		GivenΔ(pΔ1 y=((pΔ2 (t+4)×t)+((t+4)×1))) xΔ t
6.	15t,6f,7t		GivenΔ(pΔ1 y=(((pΔ2 t+ pΔ3 4)×t)+((t+4)×1))) xΔ t
7.	15t,6f...11f,12t		GivenΔ(pΔ1 y=(((pΔ2 t+0)×t)+((t+4)×1))) xΔ t
8.	15t,6f...12f,13t		GivenΔ(pΔ1 y=(((1+0)×t)+((t+4)×1))) xΔ t
9.	15t,6f...13f,14t		GivenΔ((d(y)/d(t))=(((1+0)×t)+((t+4)×1))) xΔ t
10.	15f,16t	(output)	GivenΔ((d(y)/d(t))=(((1+0)×t)+((t+4)×1)))

3.3 Example 3: Form Union and Intersection of Sets.

The Problem. Given two parenthesized lists of elements each of which represents a set, produce two more parenthesized lists which represent the union and intersection of the given sets. An empty set will be represented by an empty parenthesis pair.

The Program.

```
1.          begin string dummy s,t;   element dummy e;
2.                phrase name b;
3.   ENTER: AΔ(s) BΔ(t) → AΔ(s) BΔ(t) UΔ(s) IΔ( ) ;
4.          BΔ(...) → BΔ(bΔ...) ;
5.   LOOPB: if (...bΔ) → (...) then go to EXIT;
6.          AΔ(...) → AΔ(aΔ...) ;
7.   LOOPA: if (...aΔ) → (...) then UΔ(...) → UΔ(...b) and go to NEWB;
8.          if aΔ b → b then IΔ(...) → IΔ(...b) and go to NEWB;
9.   NEWA:  aΔ e → e aΔ and go to LOOPA;
10.  NEWB:  bΔ e → e bΔ and go to LOOPB;
11.  EXIT:  end
```

2-204

Exposition. The program assumes that the given sets are named 'A' and 'B', and it generates the union and the intersection as sets named 'U' and 'I'. In one complete execution of the program, set B is scanned once and set A is scanned once for each element in set B. Note that the identifier 'b' on Line 7 designates the element which follows the pointer 'bΔ' in the current data string (see Sec. 2.5, R2). The ellipsis, '...', represents an arbitrary string (see Sec. 2.5, R4).

Example Problems:

```
E1.  AΔ(m B28 1.3 Q) BΔ(1.3 n m)
     AΔ(m B28 1.3 Q) BΔ(1.3 n m) UΔ(m B28 1.3 Q n) IΔ(1.3 m)

E2.  AΔ( ) BΔ(1.3 n m)
     AΔ( ) BΔ(1.3 n m) UΔ(1.3 n m) IΔ( )
```

3.4 Example 4: Parenthesize.

The Problem. Given a symbol sequence enclosed in a pair of parentheses, insert parenthesis pairs into the symbol sequence so that the resulting symbol sequence is "fully parenthesized". This parenthesization will be performed according to the method of operator precedence defined in a paper by Floyd [6]. In addition to the given symbol sequence, a "precedence matrix" must be included in the initial data string. This precedence matrix is used by the programmer to specify the way in which parenthesization shall occur.

The program given here is of interest both as a device for preparing a data string for input to some other AMBIT program and as an independent exposition of a method of syntactical analysis. The automatic parenthesization of AMBIT input can be performed by other programs, including a more special program which uses a one-dimensional array to specify operator precedence and a more general program which is driven by a complete BNF grammar. The method used here appears to be an appropriate compromise for a large class of symbol manipulation problems.

2-205

The Program.

```
1.              begin segment dummy s1, s2, s3;   phrase dummy phrase;
2.                    segment name C, R;   multiple segment name L;
3.                    literal E, ERROR;
4.   ENTER:     Given∆(...) → Given∆ C∆(R∆ ... ) ;
5.   LOOP:      Given∆ C∆( ... R∆) → Given∆ ... and go to EXIT or
6.              (phrase)R∆ → phrase R∆ and go to LOOP or
7.              go to LOOKUP;
8.   RETURN:    ∃ rel∆ < and C∆ s1 ... R∆ s2 → L∆ s1 ... C∆ s2 R∆ or
9.              ∃ rel∆ = and C∆ s1 ... R∆ s2 → s1 ... C∆ s2 R∆ or
10.             ∃ rel∆ > and L∆ s1 ... R∆ s2 → C∆ s1 (...) R∆ s2 or
11.             ∃ rel∆ E  and Given∆ → Given∆ ERROR and go to EXIT;
12.             rel∆ s1 → s1 and go to LOOP;
13.  LOOKUP:    Matrix∆(((...))...) → Matrix∆(((col∆...))row∆...);
14.  ROW:       if ∃ row∆ C or ∃ row∆( ? )
15.             then row∆ s1(...) → s1(rel∆ ... ) and go to COL
16.             else row∆ s1(...) → s1(...) row∆ and go to ROW;
17.  COL:       if ∃ col∆ R or ∃ col∆( ? )
18.             then col∆ s1 → s1 and go to RETURN
19.             else col∆ s1 → s1 col∆ and
20.                  rel∆ s1 → s1 rel∆ and go to COL;
21.  EXIT:      end
```

2-206

Exposition. The initial data string must include a precedence matrix. This matrix is unchanged by the execution of the program, but controls the parenthesization which is performed by the program. The following is a precedence matrix which is typical and which is written in a form suitable for immediate incorporation into the initial data string:

MatrixΔ(	((	)	=	+	-	x	/	↑	sin	cos	d	(	(?)	))
(	(	=	<	<	<	<	<	<	<	<	<	<	<	)
=	(	>	E	<	<	<	<	<	<	<	E	<	<	)
+	(	>	>	>	>	<	<	<	<	<	E	<	<	)
-	(	>	>	>	>	<	<	<	<	<	E	<	<	)
x	(	>	>	>	>	>	E	<	<	<	E	<	<	)
/	(	>	>	>	>	E	E	<	<	<	=	<	<	)
↑	(	>	>	>	>	>	>	<	<	<	E	<	<	)
sin	(	>	>	>	>	>	>	>	E	E	E	<	E	)
cos	(	>	>	>	>	>	>	>	E	E	E	<	E	)
d	(	>	E	E	E	E	=	E	E	E	E	<	E	)
)	(	>	>	>	>	>	>	>	E	E	E	E	E	)
(?)	(	>	>	>	>	>	>	>	E	E	E	E	E	))

2-207

For any ordered pair of segments, (A,B), the precedence matrix specifies a "precedence relation", P(A,B). P(A,B) is that symbol which lies in the row labelled A and the column labelled B. If there is no row labelled A, then the row labelled '(?)' applies; and if there is no column labelled B, then the column labelled '(?)' applies. The symbol thus specified is '<', '=', '>', or 'E'.

The precedence matrix given above is only an example and may be modified to suit the requirements of the programmer. The entry for a particular precedence relation may be changed and rows and columns may be added and deleted. However, changes in the rows and columns labelled '(', ')', and '(?)' are subject to certain restrictions because of the special role of parentheses in AMBIT.

The way in which the matrix controls parenthesization can be described informally as follows: Let A and B be any two segments in the string being parenthesized such that A and B are either adjacent or are separated by a parenthesized string. Then if P(A,B) is '<', a '(' must be inserted after A; if P(A,B) is '=', no parenthesis need be inserted between A and B; if P(A,B) is '>', then a ')' must be inserted before B; and if P(A,B) is 'E', then a syntactic error in the string has been detected. These rules are constrained by the requirement that, first, parentheses must be inserted in matched, nested pairs and, second, a phrase must not be enclosed in a parenthesis pair in the final result. The second requirement excludes redundant parenthesization such as '(2.5)' or '((ALPHA-2.5))'. The string is fully parenthesized when no further parentheses can be inserted according to these rules.

In the program, 'CΔ' and 'RΔ' are used to select all possible segment pairs to which the parenthesization rules apply. 'LΔ' is used to "remember" the position of a '(' until the position of a matching ')' is determined; and 'LΔ' must be a multiple pointer because several positions for '(' may be determined before the position of a matching ')' is determined. The precedence analysis algorithm occupies Lines 4 - 12 of the program only; Lines 13 - 20 are a sub-program which places the pointer 'relΔ' in front of the precedence relation for the segment pair selected by 'CΔ' and 'RΔ'.

Example Problems. In the following examples, PM and PM* are each used to represent a symbol sequence identical to the precedence matrix which was given above, except that in PM* the entries for P(sign,sign) and P('×','×') are each changed from '>' to '=' (a total of five changes).

E1 shows the parenthesization of the equation which was the input data string of E1 for the Clear Fractions program. E2 and E3 show the effect of a change in the precedence matrix; for certain operations, such as simplification, the parenthesization of E3 is more convenient than that of E2. E4 shows the complete processing of a problem in clearing fractions,

and exhibits four data strings. The first data string in E4 is the data string supplied by the programmer; the second is the result of execution of the Parenthesize program; the third is the result of the execution of the Clear Fractions program (as per E2 of Clear Fractions); and the fourth is the result of the execution of a "De-Parenthesize" program (not given in this paper) and is the result returned to the programmer.

E1. GivenΔ(a/3+b=beta-10) PM
GivenΔ(((a/3)+b)=(beta-10)) PM

E2. GivenΔ(a×b×c+d×e×f+g×h×i=j) PM
GivenΔ(((((a×b)×c)+((d×e)×f))+((g×h)×i))=j) PM

E3. GivenΔ(a×b×c+d×e×f+g×h×i=j) PM*
GivenΔ(((a×b×c)+(d×e×f)+(g×h×i))=j) PM*

E4. GivenΔ(2.8×m1-(a/(b+2))×m2=sin(s+t)/cos(t)) PM
GivenΔ(((2.8×m1)-((a/(b+2))×m2))=((sin(s+t))/(cos(t)))) PM
ResultΔ(((((2.8×m1)×(b+2))-(a×m2))×(cos(t)))=((b+2)×(sin(s+t)))) PM
ResultΔ((2.8×m1×(b+2)-a×m2)×cos(t)=(b+2)×sin(s+t)) PM

Trace of Example Problem 1. In the following trace, the symbol '<' is used in the execution history to abbreviate " '5f', '6f', and the sub-program LOOKUP was executed and caused 'relΔ' to be positioned to the left of '<' in the precedence matrix"; and '=' and '>' have similar interpretations. The phrase '(skip operand)' is used to abbreviate the execution history '<, 8t, 12t, >, 8f, 9f, 10t, 12t, 5f, 6t', which describes the advance of 'RΔ' over an elementary operand.

1.	(input)	GivenΔ(a/3+b=beta-10) PM
2.	4t	GivenΔ CΔ (RΔ a/3+b=beta-10) PM
3.	<,8t,12t	GivenΔ LΔ1 (CΔ a RΔ /3+b=beta-10) PM
4.	>,8f,9f,10t,12t	GivenΔ CΔ ((a) RΔ /3+b=beta-10) PM
5.	5f,6t	GivenΔ CΔ (a RΔ /3+b=beta-10) PM
6.	<,8t,12t	GivenΔ LΔ1 (a CΔ / RΔ 3+b=beta-10) PM
7.	(skip operand)	GivenΔ LΔ1 (a CΔ /3 RΔ +b=beta-10) PM
8.	>,8f,9f,10t,12t	GivenΔ CΔ ((a/3) RΔ +b=beta-10) PM

```
9.   <,8t,12t              GivenΔ LΔ1 ((a/3) CΔ + RΔ b=beta-10)   PM
10.  (skip operand)        GivenΔ LΔ1 ((a/3) CΔ +b RΔ =beta-10)   PM
11.  >,8f,9f,10t,12t       GivenΔ CΔ (((a/3)+b) RΔ =beta-10)   PM
12.  <,8t,12t              GivenΔ LΔ1 (((a/3)+b) CΔ = RΔ beta-10)   PM
13.  (skip operand)        GivenΔ LΔ1 (((a/3)+b) CΔ =beta RΔ -10)   PM
14.  <,8t,12t              GivenΔ LΔ1 (((a/3)+b) LΔ2 =beta CΔ - RΔ 10)   PM
15.  (skip operand)        GivenΔ LΔ1 (((a/3)+b) LΔ2 =beta CΔ -10 RΔ )   PM
16.  >,8f,9f,10t,12t       GivenΔ LΔ1 (((a/3)+b) CΔ =(beta-10) RΔ )   PM
17.  >,8f,9f,10t,12t       GivenΔ CΔ ((((a/3)+b)=(beta-10)) RΔ )   PM
18.  5t,21t   (output)     GivenΔ(((a/3)+b)=(beta-10))   PM
```

REFERENCES.

1. Christensen, Carlos. AMBIT: a programming language for algebraic symbol manipulation. Computer Associates, Inc. (CA-64-4-R), Wakefield, Mass. October 1964.

2. Floyd, R. W. A descriptive language for symbol manipulation. J. ACM 8 (October 1961), 579-584.

3. COMIT Programmer's Reference Manual. (Second edition). MIT Press, Cambridge, Mass. 1961.

4. Farber, D. J., Griswold, R. E., and Polonsky, I. P. SNOBOL, a string manipulation language. J. ACM 11 (January 1964), 21-30.

5. Perlis, A. J., and Iturriaga, Renato. An extension to ALGOL for manipulating formulae. Comm ACM 7 (February 1964), 127-130.

6. Floyd, R. W. Syntactic analysis and operator precedence. J. ACM 10 (July 1963), 316-333.

2-210

2-211

NEW PROOFS OF OLD THEOREMS IN LOGIC AND FORMAL LINGUISTICS

by

Robert W. Floyd

2-215

CA-6505-1411
Research Paper

May 14, 1965

The research reported in this paper was sponsored in part by the Information System Theory Project under Contract AF30 (602)-3324 with the Rome Air Development Center.

New Proofs of Old Theorems in Logic and Formal Linguistics*

R. W. Floyd
CA-6505-1411
Computer Associates, Inc.
Wakefield, Massachusetts

Summary

Brief review definitions of the Post correspondence problem, semi-Thue systems, Post normal systems, and minimal linear grammars are given. Theorem 1 constructs, from any word problem in a semi-Thue system, an equivalent Post correspondence problem, showing the undecidability of the general Post correspondence problem. Theorem 2 constructs, from any Post correspondence problem, a minimal linear grammar which is ambiguous exactly if the correspondence problem has a solution, showing the undecidability of the general ambiguity problem (Other standard undecidability results on phrase structure grammars are implied). Theorem 3 constructs, from any word problem in a semi-Thue system, an ambiguity problem, combining the results of Theorem 1 and 2 by more direct means. Theorems 4 and 5 show the equivalence of the word problems for semi-Thue systems and normal systems. No new results are presented, but standard proofs have been shortened and constructions eliminated, combined, or simplified.

Definitions

The Post Correspondence Problem [8,1]

For any finite set of ordered pairs of non-null strings (c_i, d_i), the question whether there exist a number $n > 0$ and a sequence of indices $i_1, i_2, \ldots, i_n$ such that $c_{i_1} c_{i_2} \ldots c_{i_n} = d_{i_1} d_{i_2} \ldots d_{i_n}$ is the Post correspondence problem (PCP) for the given set of pairs. We define functions C and D on sequences of integers, such that if $I = i_1, i_2, \ldots, i_n$, then $C(I) = c_{i_1} c_{i_2} \ldots c_{i_n}$ and $D(I) = d_{i_1} d_{i_2} \ldots d_{i_n}$. A solution of a PCP is then a non-empty sequence of integers I such that $C(I) = D(I)$.

---

* The research reported in this paper was sponsored in part by the Information System Theory Project under Contract AF30(602)-3342 with the Rome Air Development Center.

Semi-Thue Systems [4]

A Semi-Thue system (STS) is a finite set of ordered pairs of non-null strings $\alpha_i \rightarrow \beta_i$. It is said that y is derivable from x ($x \Rightarrow y$) in a STS if there are a number $n>0$ and strings $z_1, z_2, \ldots, z_n$ such that $x=z_1$, $y=z_n$, where for each j in the range $1 \leq j < n$ there exist strings u_j and v_j and a number i_j for which $z_j = u_j \alpha_{i_j} v_j$ and $z_{j+1} = u_j \beta_i v_j$. We also say that $z_j \overset{*}{\rightarrow} z_{j+1}$. The question whether $x \Rightarrow y$ in a STS is called the word problem.

Normal Systems [7, 4]

A Normal System (NS) is a finite set of ordered pairs $\alpha_i P \rightarrow P \beta_i$ with α_i and β_i non-empty. It is said that y is derivable from x ($x \Rightarrow y$) in a NS if there are a number $n>0$ and strings $z_1, z_2, \ldots, z_n$ such that $x=z_1$, $y=z_n$, where for each j in the range $1 \leq j < n$ there exist a string u_j and a number i_j for which $z_j = \alpha_{i_j} u_j$, $z_{j+1} = u_j \beta_{i_j}$. We also say that $z_j \overset{*}{\rightarrow} z_{j+1}$. Again, the question whether $x \Rightarrow y$ in a NS is called the word problem.

Minimal Linear Grammars [6]

A minimal linear grammar (MLG) is a phrase structure grammar [1, 3] with one non-terminal character S, where each production is of the form $S \rightarrow uSv$ or $S \rightarrow u$, for terminal strings u and v. (Alternatively, it may be defined as a STS where $\alpha_i = S$ and each β_i contains at most one S. The sentences generated by the grammar are those strings y for which $S \Rightarrow y$ and y contains no occurrences of S.)

Theorem 1

For any STS Σ with word problem "$x \Rightarrow y$? " we may construct a PCP which has a solution iff $x \Rightarrow y$ in Σ.

Proof: If the productions of Σ are $\alpha_i \rightarrow \beta_i$ $(1 \leq i \leq m)$ over vocabulary $V = \{\lambda_k\}$ $(1 \leq k \leq p)$, introduce the new characters $\vdash \dashv$ $* \bar{*}$ and $\bar{\lambda}_k$ $(1 \leq k \leq p)$. Construct the PCP whose pairs are

$(*, \bar{*})$	(Π_1)
$(\bar{*}, *)$	(Π_2)
$(\lambda_k, \bar{\lambda}_k)$	$(\Pi_{3,k})$
$(\bar{\lambda}_k, \lambda_k)$	$(\Pi_{4,k})$
$(\vdash x *, \vdash)$	(Π_5)
$(\dashv, \bar{*}\ y \dashv)$	(Π_6)
$(\beta_i, \bar{\alpha}_i)$	$(\Pi_{7,i})$
$(\bar{\beta}_i, \alpha_i)$	$(\Pi_{8,i})$

where, if $w = \lambda_{k_1} \lambda_{k_2} \ldots \lambda_{k_t}$, $\bar{w} = \bar{\lambda}_{k_1} \bar{\lambda}_{k_2} \ldots \bar{\lambda}_{k_t}$. The indices symbolized by $\Pi_1, \Pi_2, \Pi_{3,1}, \Pi_{3,2}, \ldots, \Pi_{8,m}$ are assumed to be the consecutive integers from 1 to $4 + 2p + 2m$.

If, for some $n \geq 0$, $x = z_0$, $z_j \overset{*}{\rightarrow} z_{j+1}$ or $z_j = z_{j+1}$ $(0 \leq j < 2n)$, and $z_{2n} = y$ (these conditions are clearly equivalent to $x \Rightarrow y$) there is a solution I_p to the correspondence problem such that $C(I_p) = D(I_p) =$

$$\vdash z_0 * \bar{z}_1 \bar{*} z_2 * \bar{z}_3 \bar{*} \ldots \bar{z}_{2n-1} \bar{*} z_{2n} \dashv$$

To prove this, assume for any index sequences $I = i_1, i_2, \ldots, i_r$ and $J = j_1, j_2, \ldots j_s$ that I,J represents the sequence $i_1, i_2, \ldots, i_r, j_1, j_2, \ldots j_s$ and that $C(I) = c_{i_1} c_{i_2} \ldots c_{i_r}$ and $D(I) = d_{i_1} d_{i_2} \ldots d_{i_r}$. If $w = \lambda_{k_1} \lambda_{k_2} \ldots \lambda_{k_t}$, define

$$E_w = \Pi_{3,k_1} \Pi_{3,k_2} \ldots \Pi_{3,k_t} \qquad \text{and}$$

$$\bar{E}_w = \Pi_{4,k_1} \Pi_{4,k_2} \ldots \Pi_{4,k_t}, \qquad \text{so that}$$

$$C(E_w) = D(\bar{E}_w) = w$$

$$D(E_w) = C(\bar{E}_w) = \bar{w}.$$

If $z_j = u_j\ \alpha_{i_j}\ v_j$ and $z_{j+1} = u_j\ \beta_{i_j}\ v_j$, define

$$F_j = E_{u_j}\ \Pi_{7,i_j}\ E_{v_j}$$

$$\bar{F}_j = \bar{E}_{u_j}\ \Pi_{8,i_j}\ \bar{E}_{v_j}.$$

Then $C(F_j) = u_j\ \beta_{i_j}\ v_j = z_{j+1}$ and $D(F_j) = \bar{u}_j\ \bar{\alpha}_{i_j}\ \bar{v}_j = \bar{z}_j$; similarly $C(\bar{F}_j) = \bar{z}_{j+1}$ and $D(\bar{F}_j) = z_j$. On the other hand, if $z_j = z_{j+1}$, define

$$F_j = E_{z_j}$$

$$\bar{F}_j = E_{z_j}$$

so that, as above, $C(F_j) = z_j = z_{j+1}$, $D(F_j) = \bar{z}_j$, $C(\bar{F}_j) = \bar{z}_j = \bar{z}_{j+1}$, and $D(\bar{F}_j) = z_j$. Now if $I_p = \Pi_5\ \bar{F}_0\ \Pi_2\ F_1\ \Pi_1\ F_2\ \Pi_2\ F_3\ \Pi_1 \ldots \Pi_2\ F_{2n-1}\ \Pi_6$ we have $C(I_p) = \vdash x * \bar{z}_1 \bar{*} z_2 * \bar{z}_3 \bar{*} z_4 * \ldots \bar{*} z_{2n} \dashv = \vdash z_0 * \bar{z}_1 \bar{*} z_2 * \bar{z}_3 \bar{*} \ldots * \bar{z}_{2n-1} \bar{*} y \dashv = D(I_p)$. Thus $C(I_p) = D(I_p)$, and I_p is a solution of the correspondence problem.

2-220

Conversely, one sees that any solution I of the correspondence problem must begin with Π_5. In order for equal numbers of asterisks to occur in C(I) and D(I), Π_6 must occur in I, and the inital part of I through the first occurrence of Π_6 is itself a solution of the correspondence problem, because Π_6 introduces the first occurrence of "$\dashv$" in both C(I) and D(I). We may thus assume that I is of the form $\Pi_5\ I'\ \Pi_6$, where I' does not contain Π_5 or Π_6. Let $I' = G_0\ H_0\ G_1\ H_1\ G_2\ H_2 \ldots H_{n-1}\ G_n$, where the H_j are the occurrences in I' of indices Π_1 and Π_2, and the G_j are sequences of other indices. One shows by induction on j that $C(G_j) = z_{j+1}$ or $\bar{z}_{j+1}$ and $D(G_j) = z_j$ or $\bar{z}_j$, with $x = z_0$, $y = z_{n+1}$, where each z_j and z_{j+1} are of the form $z_j = v_1\ v_2 \ldots v_p$, $z_{j+1} = v'_1\ v'_2 \ldots v'_p$ with $v_k = v'_k$ or $v_k \rightarrow v'_k$, so that $z_j \Rightarrow z_{j+1}$ and $x \Rightarrow y$.

We conclude from Theorem 1 that any decision method for PCP's would yield a decision method for word problems on STS's, so that the PCP is in general undecidable[1].

Theorem 2

There is no decision procedure to determine whether a minimal linear grammar is ambiguous or not. (A grammar is said to be ambiguous if it generates some sentence in two non-trivially distinct ways) [2,3,5,6].

Proof: Given any PCP $\{(c_i, d_i)\}$ $(1 \le i \le m)$, construct the following MLG, introducing the new characters $\vdash * : \dashv$ and e_i $(1 \le i \le m)$:

$$S \rightarrow \vdash S \dashv \qquad (P_1)$$

$$S \rightarrow c_i\ S\ e_i\ * \qquad (P_{2,i})$$

$$S \rightarrow c_i : e_i\ * \qquad (P_{3,i})$$

$$S \rightarrow \vdash S\ * \dashv \qquad (P_4)$$

$$S \rightarrow d_i\ S\ *\ e_i \qquad (P_{5,i})$$

$$S \rightarrow d_i : e_i \qquad (P_{6,i})$$

Now each ambiguous sentence is (or contains a smaller sentence which is) of both the forms

$$\vdash c_{i_1}\ c_{i_2} \cdots c_{i_r} : e_{i_r}\ *\ e_{i_{r-1}}\ * \cdots *\ e_{i_1}\ * \dashv \qquad (A)$$

$$\vdash d_{i_1}\ d_{i_2} \cdots d_{i_r} : e_{i_r}\ *\ e_{i_{r-1}}\ * \cdots *\ e_{i_1}\ * \dashv \qquad (B)$$

and thus represents a solution to the PCP; conversely, each solution to the PCP generates a sentence of the form (A) using P_1, P_2, and P_3, which is also a

---

[1] Davis [4] is an accessible source for a proof that the word problem for STS's, and indeed for a particular well defined STS, is in general undecidable (recursively unsolvable).

sentence of type (B) using P_4, P_5, and P_6. By separating the productions into two distinct languages accordingly, we show the undecidability of non-empty intersection, etc. [1]. To show that an ambiguous sentence must have both forms (A) and (B), note that the innermost phrases of the sentence in both derivations must contain the only colon in the sentence, and that once the innermost phrase of a derivation is recognized, each succeeding phrase is uniquely determined by the next two characters on the right. Start from the central colon of the ambiguous string, looking at the innermost phrases in the two derivations, which must be $c_i : e_i *$ and $d_j : e_j$, for some i, working outward phrase by phrase, until one reaches $\vdash$ and $\dashv$. Details are an exercise, based on the proof of Theorem 3.

Theorem 3

For any STS Σ with productions $\{\alpha_i \rightarrow \beta_i\}$ $(1 \leq i \leq n)$ on vocabulary $V = \{\lambda_k\}$ $(1 \leq k \leq p)$ and word problem "$x \Rightarrow y$?", we may construct a minimal linear grammar. The grammar generates an ambiguous sentence exactly if $x \Rightarrow y$. (This theorem also follows directly from Theorems 1 and 2)

Proof: Consider the grammar G whose productions are

$S \rightarrow ; y : *$		(P_1)
$S \rightarrow \lambda_k S e_k *$	$(0 \leq k \leq p)$, where λ_0 is taken to be ";"	$(P_{2,k})$
$S \rightarrow \alpha_i S e_i' *$	$(1 \leq i \leq n)$	$(P_{3,i})$
$S \rightarrow \vdash S \dashv$		(P_4)
$S \rightarrow :$		(P_5)
$S \rightarrow \lambda_k S * e_k$	$(0 \leq k \leq p)$	$(P_{6,k})$
$S \rightarrow \beta_i S * e_i'$	$(1 \leq i \leq n)$	$(P_{7,i})$
$S \rightarrow \vdash x ; S * \dashv$		(P_8)

We shall show that if y is derivable from x by the sequence $x = z_1 \overset{*}{\rightarrow} z_2 \overset{*}{\rightarrow} z_3 \overset{*}{\rightarrow} \ldots \overset{*}{\rightarrow} z_n = y$, then there is an ambiguous sentence in G, of the form

$$\vdash z_1 ; z_2 ; \ldots ; z_n : * H * \dashv \tag{1}$$

for some string H, and that conversely any ambiguous sentence of G contains a phrase which is an ambiguous sentence of the form (1) with $z_j \Rightarrow z_{j+1}$ in Σ, $x=z_1$, $z_n=y$, so that if G is ambiguous, y is derivable from x. The first and last four productions constitute the subsets G_A and G_B of G; the relation $u \overset{A}{\Rightarrow} v$, for example, will mean that v is derivable from u within G_A.

For any string w over V,

$$S \overset{A}{\Rightarrow} w\, S\, E_w\, * \qquad \text{by } P_2 \tag{2}$$

$$S \overset{B}{\Rightarrow} w\, S\, *\, E_w \qquad \text{by } P_6 \tag{3}$$

where, if $w = \lambda_{k_1} \lambda_{k_2} \ldots \lambda_{k_t}$, $E_w = e_{k_t} * \ldots * e_{k_2} * e_{k_1}$.

if $z_j \overset{*}{\rightarrow} z_{j+1}$, with $z_j = u_j\, \alpha_{i_j}\, v_j$, $z_{j+1} = u_j\, \beta_{i_j}\, v_j$,

$$S \overset{A}{\Rightarrow} z_j\, S\, F_j\, * \qquad \text{by (2) and } P_{3,i_j} \tag{4}$$

$$S \overset{B}{\Rightarrow} z_{j+1}\, S\, *\, F_j \qquad \text{by (3) and } P_{7,i_j} \tag{5}$$

where $F_j = E_{v_j} * e'_{i_j} * E_{u_j}$.

If $z_1 \overset{*}{\rightarrow} z_2 \overset{*}{\rightarrow} \ldots \overset{*}{\rightarrow} z_n$,

$$S \overset{A}{\Rightarrow} z_1 ; z_2 ; \ldots ; z_{n-1}\, S\, H\, * \qquad \text{by (4) and } P_{2,0} \tag{6}$$

$$S \overset{B}{\Rightarrow} z_2 ; z_3 ; \ldots ; z_n\, S\, *\, H \qquad \text{by (5) and } P_{6,0} \tag{7}$$

where $H = F_{n-1} * e_0 * F_{n-2} * \ldots * e_0 * F_2 * e_0 * F_1$

If $x=z_1$ and $y=z_n$, then

$$S \overset{A}{\Rightarrow} \vdash z_1 ; z_2 ; \ldots ; z_{n-1} ; z_n : * H * \dashv \quad \text{by } P_4, (6), \text{ and } P_1 \tag{8}$$

$$S \overset{B}{\Rightarrow} \vdash z_1 ; z_2 ; \ldots ; z_{n-1} ; z_n : * H * \dashv \quad \text{by } P_8, (7), \text{ and } P_5 \tag{9}$$

This sentence, then, has one derivation using only productions of G_A, and one using productions of G_B; it is clearly ambiguous.

The converse is more difficult. One observes in the grammar that every sentence contains exactly one colon, and that $\vdash$ and $\dashv$ are paired and nested like parentheses. Consider any sentence with two derivations, A and B. Suppose the innermost phrases are the same in both derivations. One readily sees in G that a phrase and the two characters to its right uniquely determine the next enclosing phrase, so that the two derivations must be identical. We may then assume without loss of generality that the innermost phrases are ; y : * and : respectively. In order that the number of characters to the right of the colon not be odd in one derivation and even in the other, production P_4 must be used in one or the other. The occurrences of $\vdash$ and $\dashv$ nearest the colon are the first and last characters of a phrase in both derivations, and this phrase is an ambiguous sentence. Henceforth we only consider this sentence. An inductive argument shows that the characters between the colon and $\dashv$ alternate between * and elements of $\{e_k, e_i'\}$, with derivation A using only productions of G_A, and derivation B using only productions of G_B, and with the same number of steps in each derivation. The characters to the right of the colon uniquely determine derivations A and B. Observing the role of semicolons in G, we see that the sentence must take the form

$$\vdash z_1 ; z_2 ; \ldots ; z_n : * F_{n-1} * e_0 * \ldots * e_0 * F_2 * e_0 * F_1 * \dashv$$

where z_j is a word on V and F_j contains no e_0, and where $z_1 = x$ and $z_n = y$. In order for derivation A to generate z_1, z_1 must be of the form $z_{1,1}\, z_{1,2} \cdots z_{1,p}$

with $z_{1,j}$ either in V or in $\{\alpha_i\}$, and $F_1 = F_{1,p} * \ldots * F_{1,2} * F_{1,1}$ with $F_{1,j} = e_k$ and $z_{1,j} = \lambda_k$, or $F_{1,j} = e_i'$ and $z_{1,j} = \alpha_i$. Then, considering derivation B and F_1, one shows $z_2 = z_{2,1}\, z_{2,2} \ldots z_{2,p}$ with $z_{1,j} = z_{2,j}$ or $z_{1,j} \rightarrow z_{2,j}$, so that $z_1 \Rightarrow z_2$. One continues in this fashion, finding that $z_j \Rightarrow z_{j+1}$, so that $x \Rightarrow y$.

Theorem 4 [4, 7]

Corresponding to any semi-Thue system Σ on vocabulary $V = \{\lambda_k\}$ $(1 \leq k \leq p)$ we may construct a normal system Π such that $x \Rightarrow y$ in Σ iff $x * \Rightarrow y *$ in Π, where * is not in V and x, y are words on V.

Proof: If the productions of Σ are $\{\alpha_i \rightarrow \beta_i\}$ $(1 \leq i \leq m)$, let the productions of Π be

$$\alpha_i\, P \rightarrow P\, \beta_i \qquad \text{where } 1 \leq i \leq m \qquad (\Pi_{1,i})$$

$$* P \rightarrow P * \qquad (\Pi_2)$$

$$\lambda_k\, P \rightarrow P\, \lambda_k \qquad \text{where } 1 \leq k \leq p \qquad (\Pi_3)$$

A string of the form u * v, u and v being words on V, will be said to represent the string v u. If $x_1 \xrightarrow{*} x_2$ in Σ, then $x_1 = u\, \alpha_i\, v$, $x_2 = u\, \beta_i\, v$, and in Π,

$x_1 * = u\, \alpha_i\, v * \Rightarrow \alpha_i\, v * u$ by Π_3

$\Rightarrow v * u\, \beta_i$ by $\Pi_{1,i}$

$\Rightarrow * u\, \beta_i\, v$ by Π_3

$\Rightarrow u\, \beta_i\, v * = x_2 *$ by Π_2

so if $x \Rightarrow y$ in Σ, $x * \Rightarrow y *$ in Π. Conversely, if $x * = z_1$, $z_j \xrightarrow{*} z_{j+1}$ in Π, and $z_n = y *$, where x and y are words on V, we shall show inductively that z_j represents w_j with $w_1 = x$, $w_n = y$, and either $w_j = w_{j+1}$ or $w_j \xrightarrow{*} w_{j+1}$ in Σ.

Assume this true for j; then three cases arise:

(1) $z_j = \alpha_i u * v$, $z_{j+1} = u * v \beta_i$, so that

$w_j = v \alpha_i u$, $w_{j+1} = v \beta_i u$ and $w_j \xrightarrow{*} w_{j+1}$ in Σ.

(2) $z_j = * u$, $z_{j+1} = u *$,

$w_j = u = w_{j+1}$

(3) $z_j = \lambda u * v$, $z_{j+1} = u * v \lambda$,

$w_j = v \lambda u = w_{i+1}$

Therefore if $x * \Rightarrow y *$ in Π, $x \Rightarrow y$ in Σ.

Theorem 5

Corresponding to any Post normal system Π on vocabulary $V = \{\lambda_i\}$ $(1 \leq k \leq p)$ we may construct a semi-Thue system Σ such that $x \Rightarrow y$ in Π iff $\#x\# \Rightarrow \#y\#$ in Σ, where # is not in V and x, y are words on V.

2-226

Proof: If the productions of Π are $\{\alpha_i P \rightarrow P \beta_i\}$ $(1 \leq i \leq m)$, let the productions of Σ be

$$\#\alpha_i \rightarrow * \Delta_i \qquad (\Sigma_{1,i})$$

$$\Delta_i \lambda_k \rightarrow \lambda_k \Delta_i \quad (1 \leq k \leq p) \qquad (\Sigma_{2,i,k})$$

$$\Delta_i \# \rightarrow \nabla \beta_i \# \qquad (\Sigma_{3,i})$$

$$\lambda_k \nabla \rightarrow \nabla \lambda_k \quad (1 \leq k \leq p) \qquad (\Sigma_{4,k})$$

$$* \nabla \rightarrow \# \qquad (\Sigma_5)$$

with $\# * \Delta_i \nabla$ not in V.

A string of the form:

x # will be said to represent x

* x Δ_1 y # will be said to represent α_i x y

* x ∇ y # will be said to represent x y

where x and y are words on V. If $x_1 \xrightarrow{*} x_2$ in Π, then

$\# x_1 \# = \# \alpha_i u \# \xrightarrow{*} * \Delta_i u \#$ by $\Sigma_{1,i}$

$\Rightarrow * u \Delta_i \#$ by Σ_2

$\xrightarrow{*} * u \nabla \beta_i \#$ by $\Sigma_{3,i}$

$\Rightarrow * \nabla u \beta_i \#$ by Σ_4

$\xrightarrow{*} \# u \beta_i \# = \# x_2 \#$ by Σ_5

in Σ, so if x $\Rightarrow$ y in Π, #x# $\Rightarrow$ #y# in Σ.

Conversely, if #x# = z_1, $z_j \xrightarrow{*} z_{j+1}$ in Σ, and z_n = #y#, we shall show inductively that z_j represents w_j with w_1 = x, w_n = y, and either $w_j = w_{j+1}$ or $w_j \xrightarrow{*} w_{j+1}$ in Π. Assume this is true for j; then

(1) $z_j = \# \alpha_i u \#$, $z_{j+1} = * \Delta_i u \#$,

$w_j = \alpha_i u = w_{j+1}$

(2) $z_j = * u \Delta_i \lambda_k v \#$, $z_{j+1} = * u \lambda_k \Delta_i v \#$

$w_j = \alpha_i u \lambda_k v = w_{j+1}$

(3) $z_j = * u \Delta_i \#$, $z_{j+1} = * u \nabla \beta_i \#$,

$w_j = \alpha_i u$, $w_{j+1} = u \beta_i$, and $w_j \xrightarrow{*} w_{j+1}$ in Π.

(4) $z_j = * \; u \; \lambda_k \; \nabla \; v \; \#, \; z_{j+1} = * \; u \; \nabla \; \lambda_k \; v \; \#,$

$w_j = u \; \lambda_k \; v = w_{j+1}$.

(5) $z_j = * \; \nabla \; u \; \#, \; z_{j+1} = \# \; u \; \#,$

$w_i = u = w_{j+1}$.

Therefore if $\#x\# \Rightarrow \#y\#$ in Σ, $x \Rightarrow y$ in Π.

From the above pair of theorems, it is apparent that the general word problems for semi-Thue systems and normal systems are equivalent. In particular, the general unsolvability of either implies that of the other.

None of these theorems represents a new result; rather, each eliminates intermediate steps and constructions from the traditional proofs, shortening the path from the theory of computability to the undecidability results in formal linguistics, and making these results more accessible. The constructions used may suggest other applications to decision problems.

References

[1] Bar-Hillel, Y., et al. On Formal Properties of Simple Phrase Structure Grammars. Zeit. Phonetik, Sprachwissenschaft u. Kommunikationsforschung 14, 2 (1961) 143-172. Also in Bar-Hillel, Y., Language and Information. Addison-Wesley, Reading, Mass., 1964, 116-150.

[2] Cantor, D. G. On the Ambiguity Problem of Backus Systems. JACM 9, 4 (1962) 477-479.

[3] Chomsky, N., and Schutzenberger, M. P. The Algebraic Theory of Context-Free Languages, Computer Programming and Formal Systems. North-Holland, Amsterdam, 1963, 118-161.

[4] Davis, M., Computability and Unsolvability. McGraw-Hill, New York, 1958.

[5] Floyd, R. W. On Ambiguity in Phrase Structure Languages. Comm. ACM 5, 10 (1962) pg. 526.

[6] Greibach, S. A. The Undecidability of the Ambiguity Problem for Minimal Linear Grammars, Inf. Control 6, 2 (1963) 119-125.

[7] Post, E. L. Formal Reductions of the General Combinatorial Decision Problem, Amer. J. Math. 65 (1943) 197-215.

[8] Post, E. L. A Variant of a Recursively Unsolvable Problem. Bull. Amer. Math. Soc. 52 (1946) 264-268.

ALGORITHM 245
TREESORT 3[M1]

by

R. W. Floyd

Reprinted from the Algorithms Section of the
Communications of the ACM, Volume 7, Number 12,
December 1964.

Algorithms

G. E. FORSYTHE, J. G. HERRIOT, Editors

ALGORITHM 245
TREESORT 3 [M1]

ROBERT W. FLOYD (Recd. 22 June 1964 and 17 Aug. 1964)
Computer Associates, Inc., Wakefield, Mass.

```
procedure TREESORT 3 (M, n);
  value n; array M; integer n;
comment TREESORT 3 is a major revision of TREESORT
  [R. W. Floyd, Alg. 113, Comm. ACM 5 (Aug. 1962), 434] sug-
  gested by HEAPSORT [J. W. J. Williams, Alg. 232, Comm.
  ACM 7 (June 1964), 347] from which it differs in being an in-place
  sort. It is shorter and probably faster, requiring fewer compari-
  sons and only one division. It sorts the array M[1:n], requiring
  no more than 2 × (2↑p−2) × (p−1), or approximately 2 ×
  n × (log₂(n)−1) comparisons and half as many exchanges in
  the worst case to sort n = 2↑p − 1 items. The algorithm is
  most easily followed if M is thought of as a tree, with M[j÷2]
  the father of M[j] for 1 < j ≤ n;
begin
  procedure exchange (x,y); real x,y;
    begin real t; t := x; x := y; y := t
    end exchange;
  procedure siftup (i,n); value i, n; integer i, n;
  comment M[i] is moved upward in the subtree of M[1:n] of
    which it is the root;
  begin real copy; integer j;
    copy := M[i];
  loop: j := 2 × i;
    if j ≤ n then
    begin if j < n then
        begin if M[j+1] > M[j] then j := j + 1 end;
      if M[j] > copy then
        begin M[i] := M[j]; i := j; go to loop end
    end;
    M[i] := copy
  end siftup;
  integer i;
  for i := n÷2 step −1 until 2 do siftup (i,n);
  for i := n step −1 until 2 do
  begin siftup (1,i);
    comment M[j÷2] ≥ M[j] for 1 < j ≤ i;
    exchange (M[1], M[i]);
    comment M[i:n] is fully sorted;
  end
end TREESORT 3
```

ALGORITHM 246
GRAYCODE [Z]

J. BOOTHROYD* (Recd. 18 Nov. 1963)
English Electric-Leo Computers, Kidsgrove, Stoke-on-Trent, England
* Now at University of Tasmania, Hobart, Tasmania, Aust.

```
procedure graycode (a) dimension: (n) parity: (s); value n,s;
  Boolean array a; integer n; Boolean s;
comment elements of the Boolean array a[1:n] may together be
  considered as representing a logical vector value in the Gray
  cyclic binary-code. [See e.g. Phister, M., Jr., Logical Design of
  Digital Computers, Wiley, New York, 1958. pp. 232, 399.] This
  procedure changes one element of the array to form the next
  code value in ascending sequence if the parity parameter s
  = true or in descending sequence if s = false. The procedure
  may also be applied to the classic "rings-o-seven" puzzle [see
  K. E. Iverson, A Programming Language, p. 63, Ex. 1.5];
begin integer i,j; j := n + 1;
  for i := n step −1 until 1 do if a[i] then begin s := ¬ s;
    j := i end;
  if s then a[1] := ¬ a[1] else if j < n then a[j+1] := ¬ a[j+1]
    else a[n] := ¬ a[n]
end graycode
```

ALGORITHM 247
RADICAL-INVERSE QUASI-RANDOM POINT SEQUENCE [G5]

J. H. HALTON AND G. B. SMITH (Recd. 24 Jan. 1964 and 21 July 1964)
Brookhaven National Laboratory, Upton, N. Y., and University of Colorado, Boulder, Colo.

```
procedure QRPSH (K, N, P, Q, R, E);
integer K, N; real array P, Q; integer array R; real E;
comment This procedure computes a sequence of N quasi-
  random points lying in the K-dimensional unit hypercube
  given by 0 < xᵢ < 1, i = 1, 2, ··· , K. The ith component of
  the mth point is stored in Q[m,i]. The sequence is initiated by a
  "zero-th point" stored in P, and each component sequence is
  iteratively generated with parameter R[i]. E is a positive error-
  parameter. K, N, E, and the P[i] and R[i] for i = 1, 2, ··· , K,
  are to be given.
```

The sequence is discussed by J. H. Halton in *Num. Math.* **2** (1960), 84–90. If any integer n is written in radix-R notation as

$$n = n_m \cdots n_2 n_1 n_0 \,.\, 0 = n_0 + n_1 R + n_2 R^2 + \cdots + n_m R^m,$$

and reflected in the radical point, we obtain the R-inverse function of n, lying between 0 and 1,

$$\phi_R(n) = 0 \,.\, n_0 n_1 n_2 \cdots n_m = n_0 R^{-1} + n_1 R^{-2} + n_2 R^{-3} + \cdots + n_m R^{-m-1}.$$

The problem solved by this algorithm is that of giving a compact procedure for the addition of R^{-1}, in any radix R, to a fraction, with downward "carry".

If $P[i] = \phi_{R[i]}(s)$, as will almost always be the case in practice, with s a known integer, then $Q[m,i] = \phi_{R[i]}(s+m)$. For quasi-randomness (uniform limiting density), the integers $R[i]$ must be mutually prime.

For exact numbers, E would be infinitesimal positive. In practice, round-off errors would then cause the "carry" to be incorrectly placed, in two circumstances. Suppose that the stored number representing $\phi_R(n)$ is actually $\phi_R(n) + \Delta$. (a) If $|\Delta| \geq R^{-m-1}$, we see that the results of the algorithm become un-

A MINUTE IMPROVEMENT IN THE BOSE-NELSON SORTING PROCEDURE

2-235

by

Robert W. Floyd

The research reported here was supported in part by the Information System Theory Project under Contract AF 30 (602)-3324 with the Rome Air Development Center.

This document was submitted for publication, and later withdrawn when extensive new results in its area were discovered. Results derived by Mr. Floyd and Professor Donald Knuth (California Institute of Technology) include verification of the Bose-Nelson conjecture for $n \geq 9$. Floyd and Knuth are continuing their investigations of the problems of fixed-sequence sorting under other auspices, with the intent of publishing their results when sufficient order emerges.

A MINUTE IMPROVEMENT IN THE BOSE-NELSON SORTING PROCEDURE

ROBERT W. FLOYD
COMPUTER ASSOCIATES, INC.
Wakefield, Massachusetts

Bose and Nelson [1] conjecture that the sorting procedure which they describe requires the fewest possible comparisons and exchanges, under the constraint that each sort consist of a fixed sequence of comparisons, exchanging each pair of items compared which is not in increasing sequence. As a counterexample, consider the following method of sorting an array of twenty-one items. Consider the array as consisting of three fields F_1, F_2, and F_3, of seven items each. Sort each of the F_i separately. Merge F_1 with F_2. Now the seven smallest items lie among F_1 and F_3. Merge F_1 with F_3. Now the seven smallest items, in order, lie in F_1. Merge F_2 and F_3, completing the sort. If each of the subsidiary merges and sorts is done by the Bose-Nelson procedure, 117 comparisons and exchanges are required, as opposed to 118 for their procedure as applied to 21 items.

2-237

In general, one may observe that the Bose-Nelson procedure for merging an array of x items with an array of y items remains valid if applied to an array of x fields, each containing p items, and an array of y fields, each containing p items, replacing each comparison and exchange of two items in the original procedure by a merge of the two corresponding fields. Similarly, the Bose-Nelson procedure for sorting an array of u items remains valid if applied to an array of u fields, each initially sorted and containing p items, again replacing each comparison and exchange of the original by a merge of the corresponding fields. The proofs of validity for the original procedures need be modified only slightly for these extensions. The cost (weight) of the extended merge is clearly $\phi(x,y)\cdot\phi(p,p)$, while that of the extended sort is $q\cdot f(p)+f(q)\cdot\phi(p,p)$. The costs of the Bose-Nelson procedures are bounded below by:

$$\phi(x,y) \geq (xy)^{k/2}$$
$$f(u) \geq u^k - u,$$

where $k=\log_2 3$. The generalizations given above, although slightly improving

upon the Bose-Nelson methods for certain values of the arguments, are subject to the same lower bound. The author was unable to determine whether all methods satisfying the given constraints are so bounded.

Reference

[1] Bose, R. C. and Nelson, R. J., A Sorting Problem. J.ACM 9 (1962), 282-296.

THE SYNTAX OF PROGRAMMING LANGUAGES - A SURVEY

by

Robert W. Floyd

2-241

January, 1964

CA-64-2-R

To be published in the August, 1964 issue of
IEEE Transactions on Electronic Computers

The Syntax of Programming Languages - A Survey*

Robert W. Floyd†

Summary

The syntactic rules for many programming languages have been expressed by formal grammars, generally variants of phrase structure grammars. The syntactic analysis essential to translation of programming languages can be done entirely mechanically for such languages. Major problems remain in rendering analyzers efficient in use of space and time, and in finding fully satisfactory formal grammars for present and future programming languages.

2-242

Introduction

In recent years, few programming languages designed for widespread use have escaped having the more orderly part of their formation rules and restrictions presented in one of several simple tabular forms, somewhat like the axioms of a formal mathematical system. ALGOL, JOVIAL, FORTRAN, NELIAC, COBOL, BALGOL, MAC, APT, and their offshoots have all been defined in such a fashion (see sections A and B of the bibliography). For some of these languages, the formalism is easy and natural. For others, it is not; FORTRAN [A9] suffers needlessly, bound in the unaccustomed corsetry of her younger rival's design. Whatever the merits of formal grammars in general, some languages are best defined in words. Where formal grammars are appropriate, however, mathematical and linguistic analysis provides compilers of lowered cost and high reliability, and theoretical knowledge about the structure and value of the language itself.

Phrase Structure Grammars

The most representative and fruitful example of the use of a formal grammar in defining a programming language is the use of a phrase structure grammar to specify most of the syntactic rules of ALGOL 60 [A7, A8, B1, B5]. The form for grammatical rules used in the report which officially defines ALGOL 60 is typified [A1] by

<for statement> ::=<for clause> <statement> | <label>: <for statement>.

This assertion can be read "A for-statement is defined to be a for-clause followed by a statement, or a label followed by a colon ':' followed by a for-statement". The symbol '::=' stands for 'is defined to be'; '|' stands for 'or' and is used to separate alternative forms of the definiendum. The angular brackets '< >' are used to enclose each name of a phrase type, distinguishing it as a name, rather than the thing named. This is the reverse of the way quotation marks are used in English, for the same purpose, to distinguish

The baby can say "one word"

from

The baby can say one word.

Names of phrase types appearing in the text are hyphenated to show explicitly that the separate words of a name need have no individual meaning and that the name as a whole is used as a technical term, without such connotations as its individual words may suggest. The angular brackets enclosing a name in a grammatical rule share this function. A complete set of grammatical rules for a language written in a format equivalent to that of the example is a phrase structure grammar (PSG); a language definable by a PSG is a phrase structure language (PSL).**

In general, a phrase structure grammar, taken as a set of definitions, provides a list of alternative constructions in a definition for each syntactic type, where each construction is a list of characters and syntactic type names. A construction represents the set of phrases which can be formed by replacing each syntactic type name with a phrase of that type; the phrases of a certain type are all those represented by some construction in the definition of that type. There is usually a single syntactic type, called 'program' (or 'sentence'), which is used in the definition of no other type; the set of phrases of this type is the language defined by the grammar. On the one hand, PSG's can define some languages of considerable complexity; on the other, such simple sets of strings as that consisting of 'abc', 'aabbcc', 'aaabbbccc', etc., are demonstrably not definable by any phrase structure grammar [C4].

It is evident that a complete definition of a programming language may be expressed far more concisely by a PSG than by the corresponding English sentences, and that it is humanly impossible to read or write those sentences, with their hundreds of occurrences of 'is defined to be', 'followed by', and 'or'. If a phrase structure grammar is nearly adequate to define a language, then most of the rules defining the language can be neatly and compactly listed without explanation, conserving space, time, and clarity; attention may be concentrated on the few syntactic rules which do not fit the pattern of phrase definitions.

As mentioned above, the rules of a PSG are analogous to axioms. One who has somehow obtained a program and an understanding of its structure can use a PSG to prove the program is well formed, and to demonstrate the structure to others. The usefulness of a PSG to a programmer writing in the language, or to the compiler which translates it into machine language coding, is less apparent. A grammar does not tell us how to synthesize a specific program; it does not tell us how to analyze a particular given program [C3, pg. 48].

In order to construct programs in a phrase structure language, one may interpret every rule of its grammar as a permit to perform certain acts of substitution. Assigning a symbol to each syntactic type of a grammar, let us interpret each rule as allowing the substitution, for the definiendum, of any one of the alternative definientes. Applying these substitution rules repeatedly to the symbol designating the syntactic type 'program' or 'sentence', we arrive eventually at a sequence of symbols in which no further substitutions can take place; this string is a program or sentence in the language, the process by which it was produced being an abbreviated proof of its sentencehood. The symbols designating syntactic types, for which substitutions may be made, are called <u>non-terminal characters</u>; those undefined symbols which form sentences are the <u>terminal characters</u>.

Take, for instance, the grammar

a) <sentence> → <noun> <predicate>

b) <predicate> → <verb> <noun>

c) <noun> → John | Mary

d) <verb> → loves

Successive substitution, starting with <sentence>, gives the sequence

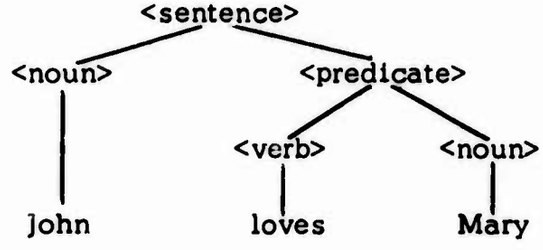
1. <sentence>
2. <noun> <predicate>
3. John <predicate>
4. John <verb> <noun>
5. John <verb> Mary
6. John loves Mary

This sequence, a derivation of the sentence "John loves Mary", is an abbreviation of the following proof:

1. Any sentence is a sentence.
   a) A noun followed by a predicate is a sentence.
2. Any noun followed by any predicate is a sentence.
   c) 'John' is a noun.
3. 'John' followed by any predicate is a sentence.
   b) A verb followed by a noun is a predicate.
4. 'John' followed by any verb followed by any noun is a sentence.
   c) 'Mary' is a noun.
5. 'John' followed by any verb followed by 'Mary' is a sentence.
   d) 'Loves' is a verb.
6. 'John loves Mary' is a sentence.

From this point of view, a sentence in a phrase structure language is the last line of a derivation from the symbol '<sentence>', provided that no further substitutions are possible [C3, Ch. 4]. The grammar is regarded not as an axiom scheme for validating sentences, but as a device for generating them. When a PSG is considered as a generative grammar, its rules are commonly called productions. The two viewpoints are substantially equivalent, but the generative viewpoint, by making explicit the process by which sentences are constructed, makes the grammar a more tractable object of study. A writer of programs in a PSL can now be thought of as a device to generate sentences, with choices between alternatives governed, for example, by the structure of a flow chart of the program. Not enough is known about linguistic behavior to specify the mechanism of choice in detail.

A compact representation of a derivation is the syntax tree [C3, G1]; the syntax tree for the derivation above is:



In general, a syntax tree is like a genealogical tree for a family whose common ancestor is <sentence>, where the immediate descendants (sons) of a symbol form one of the alternatives of the definition of that symbol, and where only the terminal characters fail to have descendants. Such a tree represents a derivation of the sentence formed by its terminal characters. It also illustrates the structure of the sentence; the terminal descendants of any node on the trees form a phrase in the sentence, of the type designated by that node. In a language satisfactorily described by its grammar, the phrases of a sentence are its meaningful units. Some compilers take advantage of this, creating a syntax tree as a structured representation of the information contained in the source program. Suitable processes then translate the tree into a computer program, or a derivation tree for an equivalent sentence in another language or a related sentence in the same language.

2-247

2-246

Syntax-Directed Analysis

A syntax-directed analyzer might be defined as any procedure capable of constructing a syntax tree for an arbitrary sentence in an arbitrary PSL. This ideal, however, is rarely achieved; most syntax-directed analyzers are restricted to languages whose grammars satisfy certain special conditions. Let us consider a typical procedure for syntax-directed analysis.

Because the analyzer makes use of a complicated hierarchy of subordinate goals in seeking its principal goal, we will introduce it with a metaphor. Suppose a man is assigned the goal of analyzing a sentence in a PSL of known grammar. He has the power to hire subordinates, assign them tasks, and fire them if they fail; they in turn have the same power. The convention will be adhered to that each man will be told only once "try to find a G" where G is a symbol of the language, and may thereafter be repeatedly told "try again" if the particular instance of a G which he finds proves unsatisfactory to his superiors. Depending on the form of the definition of G, each subordinate (S, say) should adopt an appropriate strategy:

(1) If G is a terminal character, and if it is the next character of the sentence, S must cover the character, and report success to his superior. If it is not the next character of the sentence, S must report failure. After success, if told by his superior to try again S must report failure and uncover the character.

(2) If $G \rightarrow G_1$, S must appoint a subordinate S_1 with the command, "Try to find a G_1". S repeats S_1's report to his superior, firing S_1 on a report of failure. If told to try again, S must teli S_1 to try again, again transmitting the report to his superior and firing S_1 on failure.

(3) If $G \rightarrow G_1\ G_2 \ldots . G_n$, S must appoint successively one subordinate S_i for each G_i, with the command, "Try to find a G_i". If S_i succeeds, i is increased by one, a new subordinate hired, and the process repeated until $i > n$, when S reports success. If S_i fails S_i is fired. i is decreased by one and if $i > 0$, the new S_i (predecessor of him who failed) told to try again. If $i = 0$, S reports failure, having exhausted all ways of find a G. If after success S is told to try again, he sets $i = n$, tells S_i to try again, and proceeds as before on S_i's report.

2-248

(4) If $G \rightarrow G_1 \mid G_2 \mid \ldots \mid G_n$, S must appoint successively one subordinate S_i for each G_i, with the command, "Try to find a G_i". If S_i fails he is fired, i is increased by one, a new subordinate hired, and the process repeated until $i > n$, when S reports failure. If S_i succeeds, S reports success. If after success S is told to try again, he tells S_i (who succeeded) to try again, and proceeds as before on S_i's report.

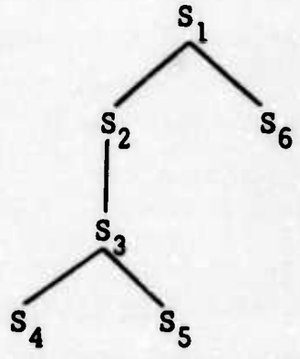
(5) All more complicated definitions can be regarded as built up from the first four types.

As an example, take the sentence 'abc' and the grammar

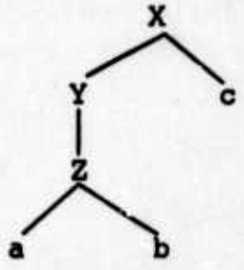
$$X \rightarrow Yc$$
$$Y \rightarrow a \mid Z$$
$$Z \rightarrow ab$$

in which X represents <sentence>. S_1 is appointed to find an X. S_1 appoints S_2 to find a Y. S_2 appoints S_3 to find 'a'. S_3 covers 'a', reports success.

S_2 reports success. S_1 appoints S_4 to find 'c'. S_4 sees 'b' in the sentence, reports failure. S_1 fires S_4 and tells S_2 to try again. S_2 tells S_3 to try again. S_3 uncovers 'a', reports failure. S_2 fires S_3, then appoints S_3 to find a Z. S_3 appoints S_4 to find 'a'. S_4 covers 'a', reports success. S_3 appoints S_5 to find 'b'. S_5 covers 'b', reports success. S_3 reports success. S_2 reports success. S_1 appoints S_6 to find 'c'. S_6 covers 'c', reports success. S_1 reports success. The organization chart of S_1 and his subordinates,



when labeled with goals rather than names, gives the syntax tree

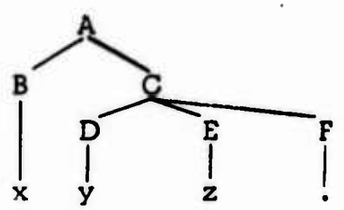


of the sentence 'abc'.

This metaphor conceals certain difficulties by relegating bookkeeping tasks to imaginary men who are assumed to automatically appear when hired, disappear when fired, remember the names of their subordinates and superiors, and so on. It is not difficult, however, by the use of a stack (pushdown list) to simulate the process on a computer, making the entire process explicit. As a convenience for the analyzer, let each definiens of the grammar be followed by the additional symbols "|-|", so that "A→B | CD" would be rewritten "A→B |CD|-|". Each subordinate in the metaphor is represented by an element S_λ of a stack,

and contains five fields: $goal_\lambda$, the fixed goal given to S_λ by his superior; i_λ, the place in the definition of goal at which S_λ is reading in the grammar; sup_λ, the name of S_λ's superior (i.e., his location in the stack); and $pred_\lambda$, the predecessor of S_λ among the subordinates of his superior. For each field, a zero specifies the absence of a value. The chief executive of the process, S_1, is set initially to have a goal of '<sentence>' with all other fields set to zero. The index λ signifies the subordinate S_λ who is currently active; the index ν signifies the first element of the stack to which no goal is currently assigned. The index j signifies the first uncovered character of the input string. The grammar is represented by the vector gram, of which each character either belongs to the language defined or is one of ($\rightarrow$, |, $\dashv$). All occurrences of S, goal, i, sup, sub, and pred, unless otherwise indexed, are implicitly indexed with λ.

When the algorithm terminates successfully, the contents of the stack represent a syntax tree for the sentence taken from the input string. Each word in the stack represents a node in the tree, where goal represents the label of the node, i is the index in gram of the '|' following the rule of the grammar applied at that node, sup designates the parent node, sub designates the rightmost son of the node, and pred designates the sibling immediately to the left of the node. Only goal, sub, and pred are needed in order to construct the tree. Thus the tree



would be represented by the stack:

	Goal	i	sup	sub	pred
1	A	?	0	4	0
2	B	?	1	3	0
3	x	0	2	0	0
4	C	?	1	9	2
5	D	?	4	6	0
6	y	0	5	0	0
7	E	?	4	8	5
8	z	0	7	0	0
9	F	?	4	10	7
10	.	0	9	0	0

2-251

Figure 1 is a flowchart for this process. There follows an item-by-item explanation of the flowchart.

2-250

A: Chief executive S_1 is appointed to find a sentence. S_2 awaits employment.

B: You are a newly appointed subordinate (S_λ); determine whether your goal is a non-terminal (defined) character, or terminal.

C,D: If the first character of the input sentence is your goal, cover it and report success to your superior; otherwise report failure and await temporary unemployment.

E: Find the beginning of the definition of your goal, by means not described here. Prepare to read that definition.

F,G,H: If you have reached a '|' in the definition of your goal, report success unless you are the chief executive, in which case you have analyzed the sentence.

I,J,K: If you have exhausted all alternatives in the definition of your goal, report failure unless you are the chief executive, in which case the input is not a sentence.

L: Otherwise, appoint a subordinate whose goal is the next character in the definition of your goal. His superior is you, his predecessor your previously junior subordinate. Remember only your most recent subordinate, protect him from other assignments, and await his report.

M: Report success to your superior, who proceeds through the definition of his goal.

N: Report failure to your superior, who will take your predecessor as his junior subordinate and fire you.

O: When told to try again, determine again whether your goal is terminal or not.

P: If terminal, uncover the input character you previously covered, and report failure.

Q,R: If goal is non-terminal, tell your junior subordinate to try again.

S,T: If you have no junior subordinate, you have exhausted an alternative; try the next one.

2-252

The serious and intrinsic flaw of the algorithm is that it fails for grammars whose rules contain certain types of cyclic formations. If the definition of A contains an alternative beginning with A, or if one of the alternatives for A begins with B, one of those for B begins with C, and one of these for C begins with A, for example, then certain choices of input string lead the procedure into an infinite loop. A grammar containing such formations is called left-recursive [G3]. It is possible, at some cost to the explanatory power of a grammar, to reformulate it excluding left-recursive definition[C11]. This has been successfully done for several programming languages in the Compass compiler [D5, D15]. A second type of syntax-directed analyzer, which is free of the left-recursion problem, constructs the syntax tree not from the top down, but from the bottom up [C11, D2, D5, D8, G4]; it is, however, as presently formulated, subject to other restrictions. All syntax-directed analyzers currently known are further restricted in practice to non-pathological languages; if a sentence is chosen at random from the grammar

S → A | B
A → (A) | (B) | x
B → (A] | (B] | y

analysis will require a time which increases exponentially with the length of the sentence, if read from left to right. At each character of the first half of the sentence, choice must be made between two alternatives. Not until the second half of the sentence is read is any information gained about the correctness of these choices. Typical processes will back up and try again many times before hitting on the right pattern. A related grammar may be designed to exhaust the patience of all known syntax-directed analyzers, whether their prejudices be left, right, or center.

It is not known whether a method of syntactic analysis is possible for which the time required for analysis does not increase exponentially with the length of the sentence, even for pathological languages. Known methods of full generality, such as the systematic generation of all sentences until a match is found, would be unacceptably slow even for short sentences. The properties of programming languages which make them legible to human readers, however, allow them to be analyzed by simple and efficient methods. A case in point is COBOL, for which a syntax-directed analyzer is greatly simplified because each choice among alternative constructions can be decided by examining the first character or word of that construction [D6].

2-252

2-253

Syntax-Controlled Analysis

An alternative approach to syntactic analysis of phrase structure languages, sometimes called syntax-controlled analysis, entails a preliminary processing of a grammar during which matrices, tables, and lists are constructed describing in some sense the possible constructions of the grammar. Analysis of sentences then makes use of these tabulations, and may even dispense entirely with the original grammar.

As an example, let us consider precedence analysis [E2], which is a formalization and extension of methods of analysis which were used in compilers even before formal grammars were employed in defining programming languages. From the grammar of ALGOL it is possible to deduce that in any ALGOL program if a left parenthesis '(' is followed by a multiplication sign '×', separated by at most one phrase, then there is some phrase to which the multiplication sign and

any phrase adjacent to it belong, but not containing the left parenthesis. The relation is symbolized, '(⋖ ×', where the sign '⋖' is read 'yields precedence to'. Similarly, if '×' is followed by '+' with at most one phrase between, some phrase contains the multiplication sign and any phrases adjacent to it, but not the plus sign. This relation is symbolized '× ⋗ +', where '⋗' is read 'takes precedence over'. We may deduce that whenever

'...(a × b + ...'

occurs in an ALGOL program, and a and b are arbitrary phrases, then 'a × b' is a phrase. A third relation, '(≐)', applies to characters of equal precedence. While analysis based on precedence relations does not yield a complete derivation of a sentence, it determines the phrases of the sentence and the operators connecting them, which is normally sufficient information for use by a compiler.

Not every grammar is amenable to precedence analysis. Yet, like phrase structure grammars, matrices representing precedence relations are generally adequate for the description of the structure of programs in standard programming languages. Because a precedence matrix can be derived from a grammar, and applied to syntactic analysis, by a completely mechanical process, precedence analysis offers much the same flexibility and universality as does syntax-directed analysis.

Neither syntax-directed nor syntax-controlled analyzers are capable, by themselves, of dealing with non-sentences. Syntax-directed analyzers are usually incapacitated by syntactic errors in their input sentences [D5, D9]. Precedence methods are more flexible, but still require explicit specification of error recovery policies. Chomsky has proposed that an adequate grammar for a natural language must account for our ability to interpret ungrammatical sentences [G3]. Such grammars are doubly necessary for programming languages, at least to the extent of localizing the effects of programming errors.

Adequacy

The phrase structure grammar, though developed as a model for natural language, is generally considered inadequate to represent either the structure or

the constraints imposed on sentences in most natural languages [C3, C5]. Nor is the PSG sufficient to fully describe the formation rules of most programming languages. Most require, for example, that the arithmetic type of each variable be declared before using it in a formula, or that dimensions of an array be specified before referring to one of its elements. Rules of this type cannot be incorporated in a PSG [C7]; nor can the rules for writing DO-loops in FORTRAN [A9]. Any rule requiring that two or more constituent phrases of a construction be identical (or different) is almost certainly beyond the scope of phrase structure definition, as is the indication of scope of nested loops by indentation.

The PSG is nonetheless a vaulable tool for describing languages, both natural and artificial. Chomsky has described it as the only theory of grammar with any linguistic motivation that is sufficiently simple to permit serious abstract study. Most published PSG's for programming languages, while not serving as complete definitions, define languages which include the programming languages as subsets satisfying simple restrictions, and correctly account for the structure of programs.

Extensions

The use of curly brackets around a part of a rule in a PSG is sometimes used to signify an arbitrary number, possibly zero, of occurrences of the form described within the brackets. As a refinement, super- and subscripts on the closing bracket, if present, signify upper and lower limits on the number. A variant uses square brackets to signify an optional single occurrence of the form described within the brackets. The COBOL syntax uses a two-dimensional display of alternatives and options. While none of these operators extends the generative power of PSG's, they all increase the convenience and explanatory power. For example, a phrase-structure description of the function f(a,b,c,d), if general enough to deal with functions of arbitrarily many variables, leads to such absurdities as assertions that 'a,b' is a phrase but 'c,d' is not. A definition using curly brackets,

<function> → <function name> (<expression>{ , <expression>})

avoids designating as phrases any parts of the function except those which serve as names or have values.

An extension to permit specification that two component phrases of a construction must be identical increases the generative power of PSG's. Such a mechanism is used in Input Language (Siberian ALGOL) [A5], to permit programs to contain relations like "$Alpha_1 \leqq \ldots \leqq Alpha_n$" but not "$Alpha_1 \leqq \ldots \leqq Beta_n$". It seems unlikely, however, that extensions will be found which, while retaining the explanatory power of PSG's, permit the complete description of even the present generation of computer languages.

Theory of Formal Languages

There exists a rapidly growing body of theory of PSG's and other formal models of language. Some of the results are of interest to the designer of compilers and the writer of programming manuals, such as the possibility of listing the allowed character pairs which may occur in programs, or the possible initial characters of each phrase type [C1, E2]. Others pertain to the design of programming languages, such as the absence of a general procedure to determine whether a PSG generates ambiguous sentences [C2, C6, C8, C10], the existence of recognizable classes of grammars which are free from ambiguity [E1, E2, E3] and the existence of languages for which all PSG's are ambiguous [C5, C14]. Chomsky [C4] and Bar-Hillel, Perles, and Shamir [C1] are important orginial papers on the general theory of PSG's; Chomsky [C5] is a thorough survey of known results about PSG's and related language-generating devices.

2-256

Unsolved Problems

Many questions of practical importance in the design of programming languages and their compilers are unanswered; some have not, to the writer's knowledge, been stated in print. It is not known, for example, how to synthesize a phrase structure grammar for a programming language, given the precedence relations of its operators. Such a synthesis method would have prevented the costly ambiguities originally present in ALGOL 60. For a given language, it is not known how to synthesize a grammar which best displays the structure of its sentences, best accommodates a particular method of syntactic analysis, or best accounts for the structure of sentences containing slight syntactic errors. It is not known whether an analyzer is possible which would not consume excessive space and time, even for pathological languages. Some of these questions are capable of precise formulation, but even rule-of-thumb solutions for any of them would be valuable.

Bibliography

The bibliography which follows includes subjects related to the syntax of programming languages insofar as they illuminate the problems of analysis and synthesis of formally defined programming languages. The bibliography is arranged by subjects, alphabetically by author within each subject. Particularly recommended as introductions to their subjects are [A7, A12, B1, C1, C3, C4, C5, D5, D12, E2, E4, F10, G1, G3] (subjects are designated by letter, individual papers by number).

Sections by subjects:

A. Formal grammars for programming languages.
B. Expositions of languages defined by formal grammars.
C. General theory of phrase structure grammars.
D. Syntax-directed analysis.
E. Syntax-controlled analysis.
F. Non-syntactic methods of analysis.
G. Related work on analysis of natural languages.
H. Miscellaneous devices useful in performing syntactic analysis.
I. Supplementary bibliographies.

A. Formal Grammars for Programming Languages

1. J. W. Backus, "The syntax and semantics of the proposed international algebraic language of the Zürich ACM-GAMM conference", Proc. Internat. Conf. Inf. Proc.; June, 1959; pp. 125-132.

2. R. Berman, J. Sharp, and L. Sturges, "Syntactical Charts of COBOL 61", Comm. ACM, vol. 5, p. 260; May, 1962.

3. R. A. Brooker and D. Morris, "A description of the Mercury Autocode in terms of a phrase structure language", Annual Review in Automatic Programming, vol. 2, Pergamon, New York, N. Y.; 1961; pp. 29-66.

4. S. A. Brown, C. E. Drayton, and B. Mittman, "A description of the APT language", Comm. ACM, vol. 6, pp. 649-658; Nov., 1963.

5. A. P. Ershov, G. I. Kohuzhin, and Yu. M. Voloshin, "Input language for a system of automatic programming", Academy of Science U.S.S.R. Computing Center, Moscow; 1961 (Russian). Academic Press, London; 1963 (English).

6. H. D. Huskey, R. Love, and N. Wirth, "A syntactic description of BC NELIAC", Comm. ACM, vol. 6, pp. 367-375; July, 1963.

7. P. Naur et al, "Report on the algorithmic language ALGOL 60", Comm. ACM, vol. 3, pp. 299-314; May, 1960.
Annual Review in Automatic Programming, vol. 2, Pergamon, New York, N. Y.; 1961; pp. 351-390. Numerische Mathematik, vol. 2, pp. 106-136; 1960.

8. P. Naur et at, "Revised report on the algorithmic language ALGOL 60", Comm. ACM, vol. 6, pp. 1-7; Jan., 1963. Numerische Mathematik, vol. 4, pp. 420-453; 1963. Computer Journal, vol. 5, pp. 349-367; Jan., 1963.

9. I. N. Rabinowitz, "Report on the algorithmic language FORTRAN II", Comm. ACM, vol. 5, pp. 327-337; June, 1962.

10. C. J. Shaw, "A specification of JOVIAL", Comm. ACM, vol. 6, pp. 721-736; Dec., 1963.

11. C. J. Shaw, "JOVIAL - a programming language for real-time command systems", Annual Review in Automatic Programming, vol. 3, Pergamon, New York, N. Y.; 1963; pp. 53-119.

12. W. Taylor, L. Turner, and R. Waychoff, "A syntactical chart of ALGOL 60", Comm. ACM, vol. 4, p. 393; Sept., 1961. (See [A7]).

13. N. Wirth, "A generalization of ALGOL", Comm. ACM, vol. 6, pp. 547-554; Sept., 1963.

14. W. W. Youden, "An analysis of ALGOL 60 syntax", Data Proc. Systems Div., Nat. Bureau of Standards, Washington, D.C.; Aug. 15, 1961. (See [A7]).

15. "Index to ALGOL 60 syntactical chart", Training and Education Dept., E.D.P., RCA, Camden, N.J.; Oct. 20, 1961. (See [A12]).

16. "COBOL 61, revised specifications for a common business-oriented language", U.S. Govt. Printing Office, Washington, D.C., O-598941; 1961.

B. Expositions of Languages Defined by Formal Grammars

1. H. Bottenbruch, "Structure and use of ALGOL 60", Inl. ACM, vol. 9, pp. 161-221; April, 1962.

2. E. W. Dijkstra, "A primer of ALGOL 60 programming", Academic Press, New York, N. Y.; 1962.

3. H. D. Huskey, M. H. Halstead, and R. McArthur, "Neliac - a dialect of ALGOL", Comm. ACM, vol. 3, pp. 463-468; Aug., 1960.

4. D. E. Knuth and J. N. Merner, "ALGOL 60 confidential", Comm. ACM, vol. 4, pp. 268- 272; June, 1961.

5. D. D. McCracken, "A guide to ALGOL programming", Wiley, New York, N. Y.; 1962.

6. D. D. McCracken, "A guide to COBOL programming", Wiley, New York, N. Y.; 1963.

7. P. Naur, "A course of ALGOL 60 programming", Regnecentralen, Copenhagen; 1961.

8. D. T. Ross, "The design and use of the APT language for automatic programming of numerically controlled machine tools", Proc. 1959 Computer Applications Symposium; pp. 80-99.

9. J. E. Sammet, "Basic elements of COBOL 61", Comm. ACM, vol. 5, pp. 237-253; May, 1962.

10. J. E. Sammet, "Detailed description of COBOL", Annual Review in Automatic Programming, vol. 2, Pergamon, New York, N. Y.; 1961; pp. 197-230.

11. H. Schwarz, "An Introduction to ALGOL", Comm. ACM, vol. 5, pp. 82-95; Feb., 1962.

12. Reference Manual, 709/7090 FORTRAN Programming System, IBM Form No. C28-6054-2.

C. General Theory of Phrase Structure Grammars

1. Y. Bar-Hillel, M. Perles, and E. Shamir, "On formal properties of simple phrase structure grammars", Applied Logic Branch, Hebrew Univ. of Jerusalem, Technical Report No. 4; 1960. Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung, vol. 14, pp. 143-172; 1961. Summarized in Comp. Rev., vol. 4, pp. 213-214; Sept.-Oct, 1963.

2. D. G. Cantor, "On the ambiguity problem of Backus systems", Jnl. ACM, vol. 9, pp. 477-479; Oct., 1962.

3. N. Chomsky, "Syntactic structures", Mouton and Co., The Hague, Netherlands; 1957.

4. N. Chomsky, "On certain formal properties of grammars", Inf. and Control, vol. 2, pp. 137-167; June, 1959. (Addendum) "A note on phrase structure grammars", Inf. and Control, vol. 2, pp. 393-395; Dec., 1959.

5. N. Chomsky, "Formal properties of grammars", Handbook of Mathematical Psychology, vol. 2, Wiley, New York, N. Y.; 1963; pp. 323-418.

6. N. Chomsky and M. P. Schützenberger, "The algebraic theory of context-free languages", Computer Programming and Formal Systems, North-Holland, Amsterdan; 1963; pp. 118-161.

7. R. W. Floyd, "On the non-existence of a phrase structure grammar for ALGOL 60", Comm. ACM, vol. 5, pp. 483-484; Sept., 1962.

8. R. W. Floyd, "On ambiguity in phrase structure languages", Comm. ACM, vol. 5, pp. 526,534, Oct., 1962.

9. S. Gorn, "Detection of generative ambiguities in context-free mechanical languages", Jnl. ACM, vol. 10, pp. 196-208; April, 1963.

10. S. A. Greibach, "The undecidability of the ambiguity problem for minimal linear grammars", Inf. and Control, vol. 6, pp. 119-125; June, 1963.

11. S. A. Greibach, "Inverses of phrase structure generators", Ph. D. thesis, Harvard, Cambridge, Mass.; June, 1963.

12. P. S. Landweber, "Three theorems on phrase structure grammars of type 1", Inf. and Control, vol. 6, pp. 131-136; June, 1963.

13. G. H. Matthews, "Discontinuity and asymmetry in phrase structure grammars", Inf. and Control, vol. 6, 137-146; June, 1963.

14. R. J. Parikh, "Language-generating devices", Res. Lab. of Electronics, MIT, Cambridge, Mass., Quarterly Progress Report, No. 60, pp. 199-212; Jan 15, 1961.

15. M. P. Schützenberger, "On context-free languages and push-down automata", Inf. and Control, vol. 6, pp. 246-264; Sept., 1963.

16. See also [E2, G3, G4].

D. Syntax-Directed Analysis

1. M. P. Barnett and R. P. Futrelle, "Syntactic analysis by digital computer", Comm. ACM, vol. 5, pp. 515-526; Oct., 1962.

2. A. L. Bastian, Jr., "A phrase-structure language translator", Air Force Cambridge Res. Labs., Hanscom Field, Mass., Report AFCRL-69-549; Aug., 1962.

3. R. A. Brooker and D. Morris, "A general translation program for phrase-structure languages", Inl. ACM, vol. 9, pp. 1-10; Jan., 1962.

4. R. A. Brooker and D. Morris, "A compiler for a self-defining phrase structure language", Univ. of Manchester, England (undated).

5. T. E. Cheatham, Jr., and K. Sattley, "Syntax-directed compiling", Proc. Spring Joint Computer Conf., vol. 25; 1964.

6. M. E. Conway, "Design of a separable transition-diagram compiler", Comm. ACM, vol. 6, pp. 396-408; July, 1963.

7. P. Z. Ingerman, "A syntax oriented compiler ...", Moore School of Elec. Engineering, Univ. of Penn., Philadelphia, Penn.; April, 1963.

8. E. T. Irons, "A syntax directed compiler for ALGOL 60", Comm. ACM, vol. 4, pp. 51-55; Jan., 1961. (See also reference [D13]).

9. E. T. Irons, "An error-correcting parse algorithm", Comm. ACM, vol. 6, pp. 669-673; Nov., 1963.

10. E. T. Irons, "The structure and use of the syntax-directed compiler", Annual Review in Automatic Programming, vol. 3, Pergamon, New York, N. Y.; 1963; pp. 207-227.

11. R. S. Ledley and J. B. Wilson, "Automatic-programming-language translation through syntactical analysis", Comm. ACM, vol. 5; pp. 145-155; March, 1962.

12. P. Lucas, "The structure of formula-translators", Mailüfterl, Vienna, Austria, ALGOL Bulletin Supplement No. 16; Sept., 1961. Elektronische Rechenanlagen; Aug., 1961.

13. B. H. Mayoh, "Irons' procedure DIAGRAM" (letter of correction), Comm. ACM, vol. 4, p. 284; June, 1961.

14. J. C. Reynolds, "A compiler and generalized translator", Applied Math. Div., Argonne Natl. Lab., Argonne, Ill. (undated).

15. S. Warshall, "A syntax-directed generator", Proc. Eastern Joint Computer Conf., vol. 20, pp. 295-305; 1961.

16. See also [C11, G4].

E. Syntax-Controlled Analysis

1. J. Eickel, M. Paul, F. L. Bauer, and K. Samelson, "A syntax-controlled generator of formal language processors", Comm. ACM, vol. 6, pp. 451-455; Aug., 1963.

2. R. W. Floyd, "Syntactic analysis and operator precedence", Jnl. ACM, vol. 10, pp. 316-333; July, 1963.

3. R. W. Floyd, "Bounded context syntactic analysis", Proc. ACM Working Conference on Mechanical Language Structures, to be published in Comm. ACM; Aug. 14, 1963.

4. R. Graham, "Bounded context translation", Proc. Spring Joint Computer Conf., vol. 25; 1964.

5. M. Paul, "ALGOL 60 processors and a processor generator", Proc. IFIP Congress; 1962; pp. 493-497.

F. Non-Syntactic Methods of Analysis

1. E. W. Dijkstra, "Making a translator for ALGOL 60", Automatic Programming Information Bulletin No. 7; May, 1961.

2. E. W. Dijkstra, "ALGOL 60 translation", Stichting Methematisch Centrum, Amsterdam, ALGOL Bulletin Supplement No. 10; Nov., 1961.

3. A. Evans, Jr., "An ALGOL 60 compiler", Computation Center, Carnegie Inst. of Technology; Aug. 27, 1963.

4. R. W. Floyd, "A descriptive language for symbol manipulation", Jnl. ACM, vol. 8, pp. 579-584; Oct., 1961.

5. A. A. Grau, "Recursive processes and ALGOL translation", Comm. ACM, vol. 4, pp. 10-15; Jan., 1961.

6. A. A. Grau, "The structure of an ALGOL translator", Oak Ridge Natl. Lab., Oak Ridge, Tenn., ORNL-3054; Feb. 9, 1961.

7. A. A. Grau, "A translator-oriented symbolic language programming language", Jnl. ACM, vol. 9, pp. 480-487; Oct., 1962.

8. P. Naur, "The design of the GEIR ALGOL compiler", Part I, BIT, vol. 3, p. 124, 1963.

9. D. T. Ross, "An algorithmic theory of language", Electronic Systems Lab., MIT, Cambridge, Mass., ESL-TM-156; Nov., 1962.

10. K. Samelson, "Programming languages and their processing", Prcc. IFIP Congress; 1962; pp. 487-492.

11. K. Samelson and F. L. Bauer, "Sequential formula translation", Comm. ACM, vol. 3, pp. 76-83; Feb., 1960.

2-264

G. Related Work on Analysis of Natural Languages

1. D. G. Bobrow, "Syntactic analysis of English by computer- a survey", Proc. Fall Joint Computer Conf., vol. 24, pp. 365-387; 1963.

2. T. E. Cheatham, Jr. and S. Warshall, "Translation of retrieval requests couched in 'semi-formal' English-like language", Comm. ACM, vol. 5, pp. 34-39; Jan., 1962.

3. N. Chomsky and G. A. Miller, "Introduction to the formal analysis of natural languages", Handbook of Mathematical Psychology, vol. 2, Wiley, New York, N. Y.; 1963; pp. 269-322.

4. S. Kuno and A. G. Oettinger, "Multiple-path syntactic analyzer", Proc. IFIP Congress; 1962; pp. 306-312.

5. See also [C3].

2-265

H. Miscellaneous Devices Useful in Performing Syntactic Analysis

1. R. W. Floyd, "Ancestor" (Algorithm 96), Comm. ACM, vol. 5, pp. 344-345; June, 1962

2. A. W. Holt, "A mathematical and applied investigation of tree structures for computer syntactic analysis", PH. D. thesis, Univ. of Penn., Philadelphia, Penn.; 1963.

3. S. Warshall, "A theorem on Boolean matrices", Jnl. ACM, vol. 9, pp. 11-12; Jan., 1962.

I. Supplementary Bibliographies

1. R. A. Kirsch, "The application of automata theory to problems in information retrieval (with selected bibliography)", National Bureau of Standards, Washington, D.C., Report 7882; March 1, 1963.

2. O. Kesner, "Bibliography: ALGOL references", Comp. Revs., vol. 3, pp. 37-38; Jan.-Feb., 1962.

3. U. M. Voloshin, "Bibliography on automatic programming", Institut Matematiki Sibirskogo Otdeleniia Akademii Nauk S.S.S.R., Novosibirsk; 1961.

4. V. H. Yngve et al, "Towards better documentation of programming languages (ALGOL 60, COBOL, COMIT, FORTRAN, IPL-V, JOVIAL, NELIAC), Comm. ACM, vol. 6, pp. 76-92; March, 1963.

5. W. W. Youden, "Index to the Communications of the ACM volumes 1 - 5, 1958-1962", Comm. ACM, vol. 6, pp. 11-32; March, 1963.

6. "ALGOL references in the Communications of the ACM, 1960-1961", Comm. ACM, vol. 4, p. 404; Sept., 1961.

7. "Automatic programming - a short bibliography", Annual Review in Automatic Programming, vol. 1, Pergamon, New York, N. Y.; 1960; pp. 291-294.

8. See also [C5, C6, E1, E5, F9, F10, G1, G3], which contain extensive bibliographies relevant to their subjects.

Footnotes

* Received

† Computer Associates, Inc., Wakefield, Massachusetts

** The type of grammar described here is sometimes called a context-free phrase structure grammar, as distinguished from a more general type of grammar, the context-dependent phrase structure grammar. The latter has no known applications to programming languages, the term "phrase structure" is not necessarily appropriate for a context-dependent grammar, and the term "context-free" has certain misleading implications; we will therefore use the short term "phrase structure grammar" for what is sometimes also called a context-free phrase structure grammar [C6], simple phrase structure grammar [C1], or Type 2 grammar [C4].

FLOWCHART LEVELS

2-271

by

Robert W. Floyd

(Preliminary Draft)

The work reported here was supported by the Information System Theory Project under Contract AF 30 (602)-3324 with the Rome Air Development Center.

Flowchart Levels (Preliminary Draft)

It is the intention of this paper to suggest a precise interpretation of the intuitive notion of levels within a program or flowchart (e.g., "inner" and "outer" loops), to demonstrate some properties of the proposed formalism, and to present algorithms to classify a program into levels as a preliminary to optimization, segmentation, etc.

Terminology:

A flowchart F is a finite set of vertices (singular:vertex) and edges. Each edge in F is an ordered pair (v_1, v_2) of vertices in F. If $e=(v_1, v_2)$ is an edge, we say that v_1 is the tail $t(e)$ of e and that v_2 is the head $h(e)$ of e. A path through a flowchart is a sequence of edges $e_1, e_2, \ldots, e_p$ $(p\geq 1)$ such that for $1\leq j<p$, $h(e_j)=t(e_{j+1})$. A loop is a path such that $h(e_p)=t(e_1)$. We say $v_1 \rightarrow v_2$ if there is an edge (v_1, v_2). We say $v_1 \Rightarrow v_2$ if there is a path such that $t(e_1)=v_1$, $h(e_p)=v_2$. We say $v_1 \Leftrightarrow v_2$ if $v_1 \Rightarrow v_2$ and $v_2 \Rightarrow v_1$. Clearly "$\Rightarrow$" is transitive, and "$\Leftrightarrow$" is both commutative and transitive. A subset S of F is closed if it contains the head and tail of each of its edges. The closure of a subset is the union of the set with the heads and tails of its edges. Clearly closure(S) is closed, closure(closure(S))=closure(S), and closure(S)=S if and only if S is closed.

We say that an edge is an exit of a loop if its tail is a head (or, equivalently, a tail) of some edge in the loop. We assume that each flowchart has at least one vertex (a start) which is not the head of any edge, and at least one (a finish) which is not the tail of any edge, and that each vertex lies on a path from a start to a finish.

We now recursively define the level $l(x)$ of an element x of a flowchart.

(1) If e is an edge belonging to no loops, $l(e)=0$. That is, if $e=(v_1, v_2)$, $l(e)=0$ if $v_2 \not\Rightarrow v_1$.

(2) For $\lambda > 0$, if every loop to which e belongs has an exit e' with $l(e')<\lambda$, and if $l(e)$ is not less than λ, then $l(e)=\lambda$.

One sees that if there are no edges on level λ, there are also none on level $\lambda+1$, since any edge e which would satisfy the requirement for $l(e)=\lambda+1$ would also satisfy that for $l(e)=\lambda$.

We define the level of a vertex v to be the maximum level of the edges leading into v, $\max_{\lambda} (l(e))$, $v=\text{head}(e)$. We take 0 to be the maximum if no such edges exist. We define F_λ to be the subset of F consisting of vertices v and edges e such that $l(v)\geq\lambda$, $l(e)\geq\lambda$.

Theorem 1. Each edge on level $\lambda>0$ belongs to a loop in F_λ.

Proof: If $l(e)=\lambda>0$, let us assume that e belongs to no loop in F_λ. Then any loop L containing e contains some edge, say e', with $0<l(e')=\lambda'<\lambda$. By definition, every loop (including L) containing e' then has an exit e'' such that $l(e'')=\lambda''<\lambda'$, so that $\lambda''\leq\lambda'-1<\lambda-1$. Every loop through e then has an exit whose level is less than $\lambda-1$; if $l(e)$ is not less than $\lambda-1$, it satisfies the defining condition that $l(e)=\lambda-1$, contrary to hypothesis. Therefore, e must belong to some loop in F_λ.

Theorem 2. For any vertex v, the maximum level of the edges whose head is v is the same as the maximum level of the edges whose tail is v.

Proof: Suppose, on the contrary, that $e=(v_1, v)$, $l(e)=\lambda$, and the levels of all edges whose tail is v are less than λ. (Thus $\lambda>0$). By Theorem 1, e

belongs to a loop in F_λ, and this loop must contain an edge e' whose tail is v, with $l(e') \geq \lambda$, contrary to hypothesis. By this and the symmetric argument, we show that, since neither of the two maxima is greater, they must be equal.

Corollary: The level of a vertex v is $\max_{v=\text{tail}(e)} (l(e))$

Theorem 3. F_λ is closed.

Proof: For $\lambda=0$, $F_\lambda=F$, closed by definition. Otherwise, if $e=(v_1, v_2) \in F_\lambda$, $l(e) \geq \lambda$, and $l(v_1) \geq l(e)$, $l(v_2) \geq l(e)$, so v_1 and v_2 belong to F_λ.

Theorem 4. Every edge has a level.

Proof: By assumption, every edge lies on a path from a start to a finish. If the edges of this path are $e_p, e_{p-1}, \ldots, e_0$ we show $l(e_i) \leq i$. We proceed by induction on i. For i=0, the head of e_0 is a finish, so e_0 belongs to no loop, and $l(e_0)=0$. For $i>0$, $l(e_{i-1}) \leq i-1 < i$, and e_{i-1} is an exit of any loop containing e_i, so $l(e_i) \leq i$.

Define $\xrightarrow{\lambda}$, $\overset{\lambda}{\Rightarrow}$, and $\overset{\lambda}{\Leftrightarrow}$ to have the same meaning in F_λ that their counterparts have in F.

Theorem 5. If $v \overset{\lambda}{\Rightarrow} v'$, then $v' \overset{\lambda}{\Rightarrow} v$, for $\lambda > 0$.

Proof: If $v_1 \xrightarrow{\lambda} v_2$, then by Th. 1, $v_2 \overset{\lambda}{\Rightarrow} v_1$. If $v \overset{\lambda}{\Rightarrow} v'$, there is a path in F_λ from v to v', and $v=v_1 \xrightarrow{\lambda} v_2 \xrightarrow{\lambda} v_3 \xrightarrow{\lambda} \ldots \xrightarrow{\lambda} v_p=v'$, so $v'=v_p \overset{\lambda}{\Rightarrow} \ldots \overset{\lambda}{\Rightarrow} v_3 \overset{\lambda}{\Rightarrow} v_2 \overset{\lambda}{\Rightarrow} v_1=v$. By the transitive property, $v' \overset{\lambda}{\Rightarrow} v$, and $v \overset{\lambda}{\Leftrightarrow} v'$.

Corollary: Any path in F_λ ($\lambda>0$) belongs to a loop in F_λ.

Theorem 6. If there is a path P from vertex v to a finish, and if k of the vertices of the path are tails of more than one edge, $l(v) \leq k$

Proof: An exercise for the reader, using Theorem 4 and the Corollary of Th. 2.

The algorithm to assign levels to vertices and edges can be described as follows:

1. Label each vertex and edge with 0 if it belongs to no loops. Set $\lambda=0$.
2. Increase λ by 1.
3. Label each unlabeled vertex with λ if it is a tail of an edge labeled $\lambda-1$. If no such vertices exist, stop.
4. Label each unlabeled vertex or edge with λ if it belongs to no loop whose vertices are unlabeled.
5. Return to step 2.

Any nodes or edges left unlabeled by this process belong to no paths which reach a finish, and thus belong to non-terminating loops.

The process of finding paths and loops in a graph or subgraph F on n vertices can be set up as follows:

1. Initialize the nxn square array M to zeroes.
2. For each edge (v_i, v_j), set M_{ij} to 1. Now M represents "$v_i \rightarrow v_j$".
3. Create a copy M* of M

5. for j=1 to n
for i=1 to n
if $M^*_{ij}=1$
for k=1 to n
if $M^*_{jk}=1$, $M^*_{ik} \leftarrow 1$
Now M* represents $v_i \Rightarrow v_j$

6. for i=1 to n
for j=1 to n
$M^{**}_{ij} \leftarrow M^* \times M^*_{ji}$
Now M** represents $v_i \Leftrightarrow v_j$

7. for i=1 to n
$V_i \leftarrow M^*_{ii}$
Now V represents "v_i belongs to a loop"

2-277

In the absence of detailed information about exit probabilities, it is plausible to attempt to improve the speed of a program by assigning high speed storage, index registers, etc., principally to nodes and edges on the program's highest levels, and by segmenting the program and changing storage

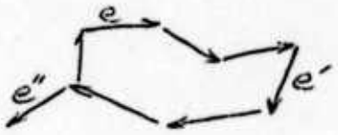
and register allocations, where possible, along edges on the lowest levels.

I am indebted to Dr. Donald Knuth for suggesting the problem to which the prededing is a partial solution.

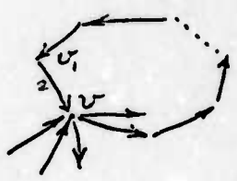
Robert W. Floyd
Computer Associates, Inc.
Wakefield, Mass.

Illustrations

Theorem 1:

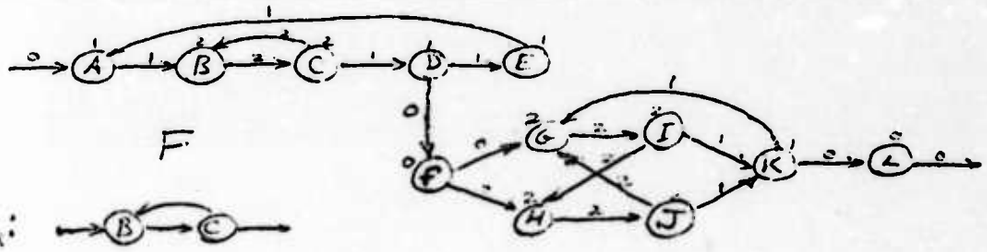


Theorem 2:



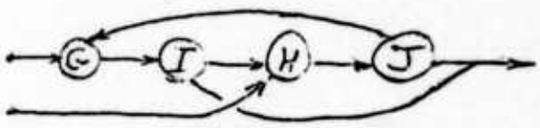
Examples:

2-279



$F_{2\alpha}$:

$F_{2\beta}$:

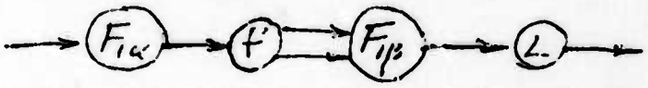


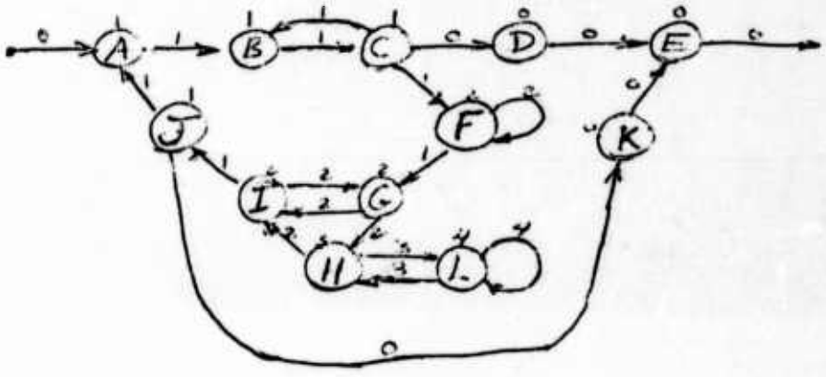
(Note that $F_{2\beta}$ has two distinct entrances)

$F_{1\alpha}$:

$F_{1\beta}$:

F_0:





2-278

NON-DETERMINISTIC ALGORITHMS

2-283

by

R. W. Floyd

(Preliminary Draft)

The work reported here was supported by the Information System Theory Project under Contract AF 30 (602)-3342 with the Rome Air Development Center.

NON-DETERMINISTIC ALGORITHMS

(Preliminary Draft)
R. W. Floyd

There is a class of algorithms in which at intervals choices must be made in a seemingly arbitrary manner; not until later in the execution of such an algorithm does one learn whether or not the sequence of choices made was correct. This class includes several algorithms for syntactic analysis of formal languages (e.g. both varieties of syntax-directed analyzer [ ], and the predictive analyzer [ ]). It includes algorithms for solving certain problems in the theory of sorting [ ]. It includes areas of cryptanalysis, game playing, theorem proving, and certain aspects of operations research such as the traveling salesman problem. This list is by no means exhaustive.

Normally a process of retracing or "backtracking" is built into such algorithms. It is our purpose to show that an algorithm in this area may be designed without specifying the backtrack process, thereby eliminating much of the tedious detail and simplifying the structure. In fact, the algorithm may be designed as if it were to be executed by a person or machine which, by insight or magic, always makes a correct sequence of choices. The algorithm is then expanded by mechanical means into a larger and more complicated process executable by machines blessed with no more insight or magic than present-day digital computers possess.

To be specific, let us consider algorithms to be defined by flow charts, i.e. directed graphs with labeled edges (arrows) where the nodes (boxes) contain operations of a few designated types; in particular:

(1) Assigning the value of an expression to a variable.

(2) Determining whether a proposition is true or false, and choosing one of two exit edges accordingly.

(3) Writing output information.

(4) Reading input information.

(5) Starting (ie., marking the point where the process begins).

(6) Halting (ie., marking a point where the process ends).

(7) Calling a named subroutine.

(8) Starting a named subroutine (ie., marking its entry).

(9) Halting a named subroutine (ie., marking its exit; for simplicity we assume it to have one entry and one exit).

We assume all nodes to have one or, in Case (2), two exits. To simplify later developments, we adopt the convention that each node has only one entry, except those of type:

(10) Null operation (ie., joining several paths in the flow-chart).

A non-deterministic algorithm (NDA) is an algorithm extended to include the operation of:

(11) Choosing an arbitrary integer in the range from 1 to n, for specified n, and assigning the chosen integer to a specified variable. This operation will be written $x \leftarrow \text{choice}(n)$.

where it is further required that each halt in the algorithm be marked as a success or a failure. A computation of a non-deterministic algorithm is a path through its flowchart, executing the indicated operations at each node on the way, which halts at a node marked as a success. An input sequence (or set of initial values of variables, or both) is said to be accepted by an NDA if it is accepted by some computation of that NDA. An output of an NDA is the output of some computation. If there is only one computation, we may speak of the output.

For a fixed input sequence and set of initial values, a NDA may have any number of computations, including zero and infinity. One may show, however, that if every path through the flow chart terminates, the NDA has a finite number of computations. Depending upon circumstances, one may be interested in finding one or all of these computations and their outputs.

We shall demonstrate the method of construction for a NDA of a corresponding (deterministic) algorithm (DA) which will write one or all of the output sequences of the NDA, provided that all paths through the NDA terminate. For each node of the NDA, with specified labels on its entries and exists, we construct one or more nodes for the DA, with specified labels. When these constructed nodes are connected as their labels indicate, they form a DA equivalent to the NDA in the sense specified above. In Table 1, the first column shows a node of the NDA; the second and third show corresponding nodes of the DA. Roughly speaking, nodes in the second column perform the same operations as those in the first, but additionally save some information on stacks, and nodes in the third column undo the effects of corresponding nodes in the second. The process uses three auxiliary stacks M, R, and W (for memory, read, and write) and an auxiliary variable T. For each subroutine S an alternate entry S' is constructed which communicates freely with S, and uses the same exit cell.

Short explanations of Table 1 follow:

(1) Before assigning the value of an expression f to a variable X, the previous value of X is saved on the memory stack M. It is restored in backtracking.

(1a) If the assignment to X is a reversible transformation (e.g., X←X+1) X need not be saved. In backtracking, the inverse transformation (e.g., X←X-1) is applied.

(2) Branches of control discard no information, and thus require no stack references. In backtracking, a branch becomes a join.

(3) Information to be written is accumulated on the write stack W until a success is reached.

(4) The use of the read stack R simulates the usually impossible backing up of an input device. R is initially empty.

(5) When the backtrack process returns to the start, all choices have been tried and the process halts.

(6a) When a success termination is reached, accumulated output is written. The process either halts, or if all computations are desired, begins backtracking.

(6b) Failure initiates backtracking in search of the most recent choice not all of whose alternatives have been explored.

(7,8,9) Upon return from a subroutine S, T=1 if the subroutine succeeded; T=0 if all paths resulted in failure. The backtrack entry S' of S effectively asks for an as yet unexplored alternate computation of S. It is assumed that S and S' are simply alternate entries to a single subroutine with a single exit cell.

(10) Before paths of control join, enough information is stored on the memory stack M to permit the backtrack process to select the proper path.

(10a) If at such a point some proposition P in the flowchart variables is sufficient to determine which entry was taken, the stack need not be used.

(11) After saving the initial value of X, X is assigned the value of 1. If all subsequent paths fail, X is incremented until all values $1 \leq X \leq n$ have been explored, at which time the initial value of X is restored, and backtracking continues.

2-289

Among the other ways of implementing NDA's, two have frequently been used. The first saves on a stack the entire state of all variables whenever a choice is made (See [ ]). A failure termination causes unstacking until a choice with unexplored alternatives is found. The second pursues alternate paths in parallel through some simulation of multiple processors (See [ ]). Both may make excessive memory requirements, and would require extensions to deal with input and output. A third would use hardware designed specifically for NDA's. All operations would be reversible under control of a "backtrack" switch, set by any failure halt. Such a machine might be built or simulated.

After expansion of an NDA into a DA, standard economization techniques may be fruitfully applied to the resulting flowchart. In addition, certain rules apply particularly to NDA's. It is not necessary to provide for backup or stacking in the initialization portion of an NDA, executed only before encountering any choices. It is not necessary to stack any variable whose value at the particular node is determinate from other variables.

It will be found that loop-free NDA's correspond to DA's with nested loops; NDA's containing loops correspond to DA's with iterated (i.e. variably nested; see [ ]) loops; and that NDA's containing recursive subroutines correspond

to DA's containing recursive coroutines (see [ ]). The desirability of discovering failure conditions as early as possible must be apparent.

On machines having discs, or other large auxiliary memories, the inactive ends of the stacks used in simulating NDA's can be moved out of fast memory when memory space is scarce.

The possibility should be considered of preserving certain information from being erased when backtracking. For example, if each solution to a problem has an associated cost, we might save the cost for the most recent solution, and accept only those subsequent solutions of lower cost. This extension would be useful for certain problems of sorting and operations research, for example.

Example

The flowchart of Figure 2 represents a non-deterministic algorithm for top-down syntax-directed analysis. It is assumed that the input is to be parsed as a phrase of type root. When the input string has been exhausted, we assume that the next character read is "$\dashv$", essentially an end-of-file symbol. We assume that for each non-terminal character X, n_X is the number of productions $X \rightarrow y_1$, $X \rightarrow y_2$, . . ., $X \rightarrow y_{n_X}$ and that $y_i \dashv$ is stored in the array p, starting at p_j where $j = prod_{X,i}$. GOAL is the input stack for the subroutine RECOGNIZER, whose task is to read a phrase of the type named by the top word of GOAL and then to unstack that word. Since RECOGNIZER is a recursive subroutine, there are additional implicit stacking operations for i and j whenever it is called.

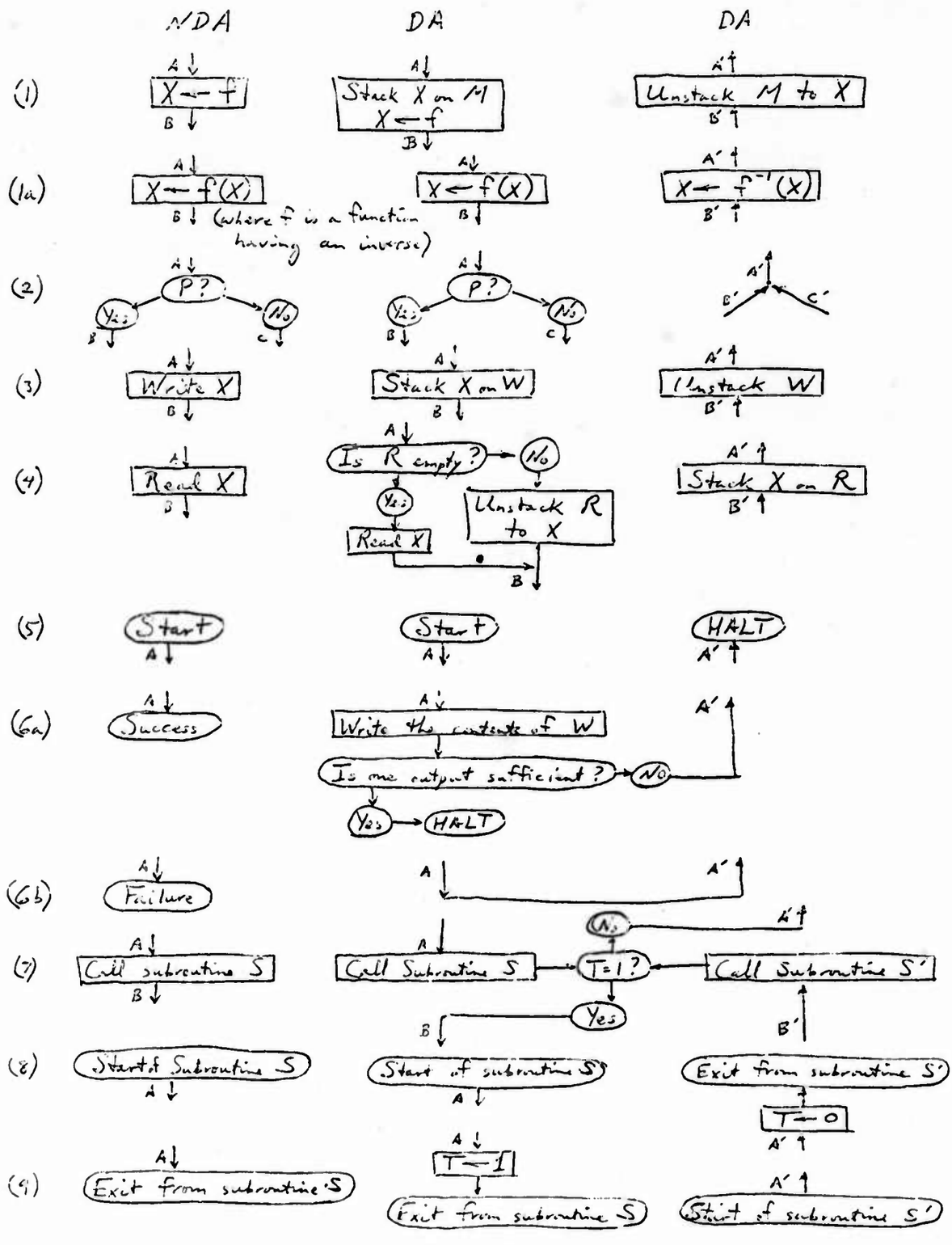

NDA
DA
DA
(1)
X ← f
Stack X on M
X ← f
Unstack M to X
(1a)
X ← f(X)
X ← f(X)
X ← f⁻¹(X)
(where f is a function having an inverse)
(2)
P?
Yes
No
(3)
Write X
Stack X on W
Unstack W
(4)
Read X
Is R empty?
No
Yes
Read X
Unstack R to X
Stack X on R
(5)
Start
Start
HALT
(6a)
Success
Write the contents of W
Is the output sufficient?
No
Yes
HALT
(6b)
Failure
(7)
Call subroutine S
Call Subroutine S
T=1?
Call Subroutine S'
Yes
No
(8)
Start of Subroutine S
Start of subroutine S
Exit from subroutine S'
T ← 0
(9)
Exit from subroutine S
T ← 1
Exit from subroutine S
Start of subroutine S'

TABLE 1

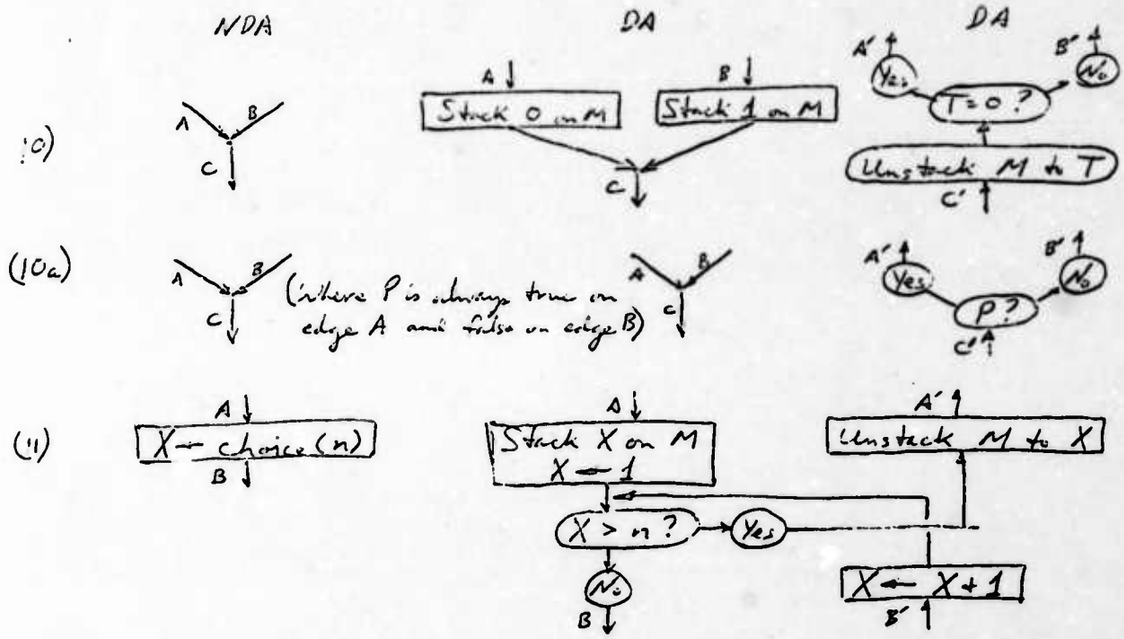


TABLE 1 (Continued)

Example

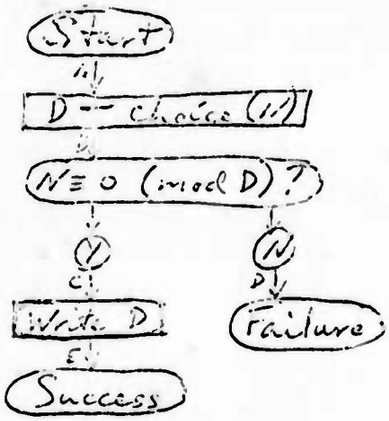


Figure 1a

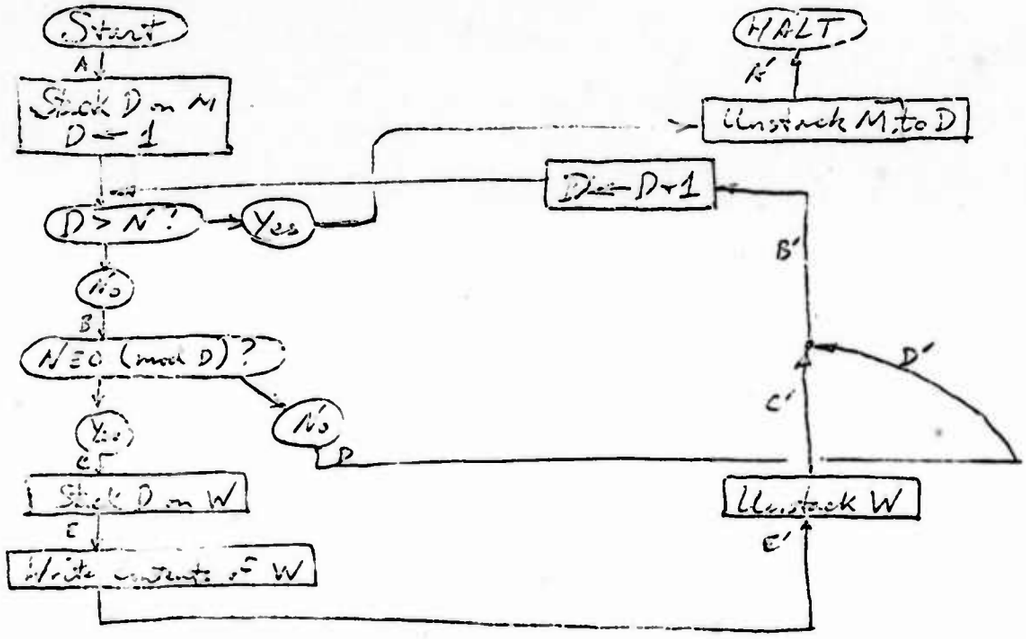


Figure 1b

Figure 1a represents a NDA to print some divisor of the integer N'. Figure 1b is a corresponding DA, constructed with the aid of Table 1, to print all divisors of N. The mechanical production of Figure 1b has resulted in obvious inefficiencies which one could easily remove.

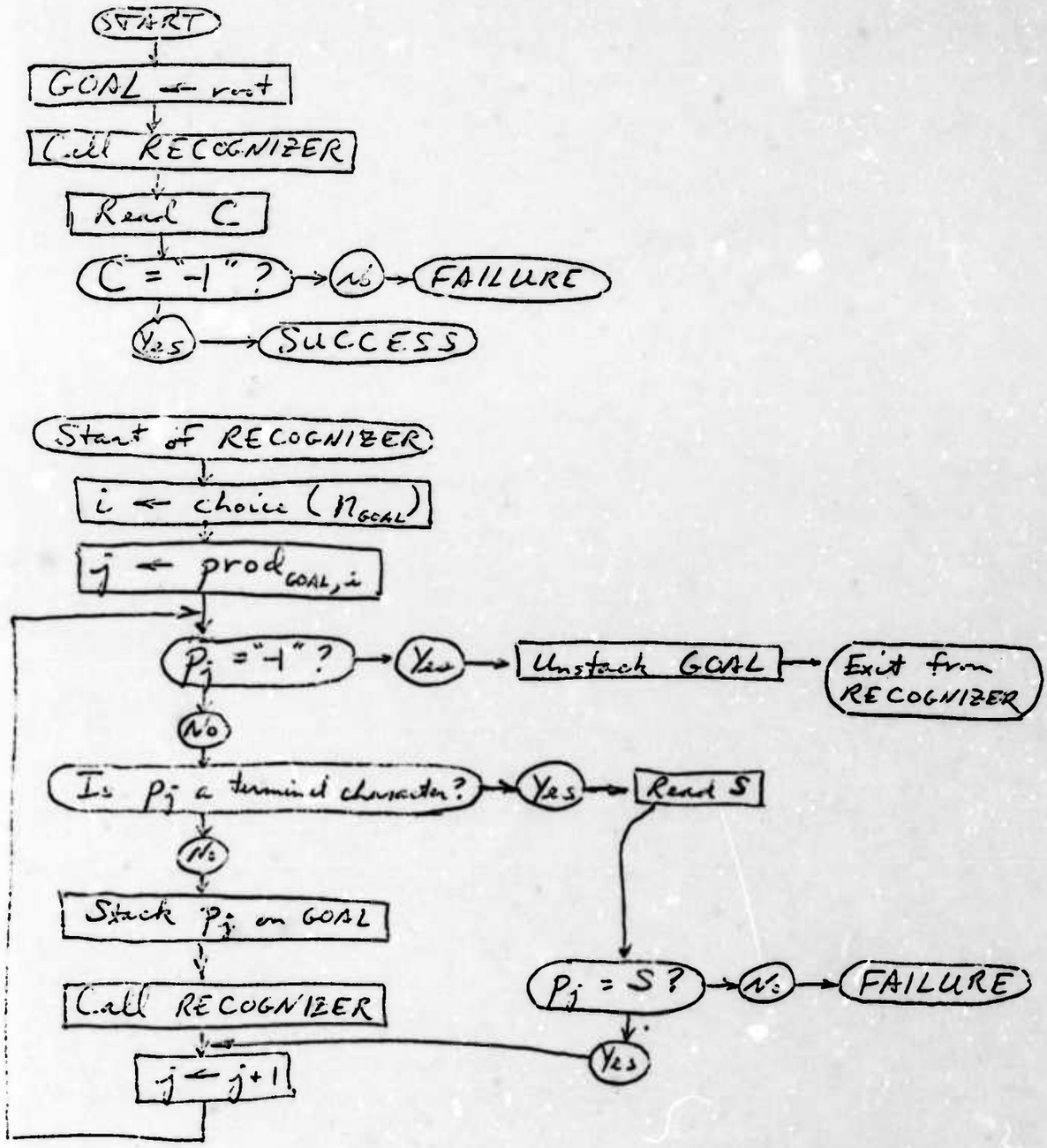

START
GOAL ← root
Call RECOGNIZER
Read C
C = "-1" ?
No
FAILURE
Yes
SUCCESS
Start of RECOGNIZER
i ← choice (n_GOAL)
j ← prod_GOAL,i
P_j = "-1" ?
Yes
Unstack GOAL
Exit from RECOGNIZER
No
Is P_j a terminal character?
Yes
Read S
No
Stack P_j on GOAL
Call RECOGNIZER
P_j = S ?
No
FAILURE
Yes
j ← j+1

Figure 2

AN ENVIRONMENT FOR AN OPERATING SYSTEM

by

G. F. LEONARD & J. R. GOODROE

Computer Associates, Inc.

Wakefield, Massachusetts

2-299

Reprinted with permission from the 1964 Proceedings of the
Association for Computing Machinery
New York, New York

AN ENVIRONMENT FOR AN OPERATING SYSTEM*

G. F. Leonard & J. R. Goodroe
Computer Associates, Inc.
Wakefield, Massachusetts

Conventional operating systems are limited in scope and are designed to deal with very specific problems in the utilization of a computer facility. As more sophisticated programming techniques and new applications for computers are developed, it becomes increasingly apparent that operating systems, as currently conceived, do not adequately cope with the resulting problems.

In this paper, a new approach to computer facility utilization is proposed which is based on the concept of extending the operations of a computer with software so as to provide a proper environment for an operating system. The organization and functional descriptions of a system, called an "extensible machine", is presented with the advantages it affords over conventional systems being explicitly stated. Criteria for developing such a system are presented and the capabilities of the system are discussed in the light of current problem areas such as parallel processing, real time processing, and intelligence systems.

Introduction

In the past, computers were used in a very straightforward manner. Jobs to be run on a computer were composed of programs, units capable of independent execution. Programs were written to utilize a specific subset of the equipment inventory available for processing and were not usually easily adaptable to configuration changes (e.g., the loss of a "printer" because of equipment malfunction). The order of execution among programs was generally controlled by the operators of the equipment, often in a somewhat arbitrary manner. It was incumbent on the operators of the computer to keep their records continually updated as to the "current" version of input files, to properly label outputs, and to run the programs of a job in the proper sequence.

As computers and computer users became more sophisticated, operators were no longer able to fulfill this role rapidly and reliably. This resulted in inefficient utilization of the equipment and a lack of responsiveness to user priorities. It was soon noted that many of the tasks performed by the operators were more suited to execution by computers than by humans. This led to the development of computer programs, called "monitors" and "operating systems", to automatically handle the scheduling and execution of user jobs. As other techniques were developed for simplifying the problems of preparing and executing user programs (e.g., assemblers, compilers, data packages), it was recognized that these programs could be employed advantageously as extensions to the operating system. This was the first step toward the development of an "extended computer" [1], an environment in which programs are written, debugged, maintained, and executed.

Conventional operating systems are purported to be systems which provide for the efficient and timely execution of programs on a computer. It seems to be the consensus of opinion that, when used in conjunction with such programming aids as compilers, assemblers, and data handling packages, an operating system is the answer to the problems attendant to utilizing a large computer facility. Modern systems allocate machine components, such as magnetic tapes, to the programs as they are loaded, inform the operator of requirements for mounting or dismounting the specific tapes required by the current job (or, in some cases, later jobs), load the programs, provide for the linkage of the programs to the computer components allocated, and transfer control to the programs. When a program terminates, it returns control to the system which then proceeds to ready the next job, and so on.

While the benefits enjoyed from such schemes are apparent and considerable, such systems have certainly not solved all of the problems. In particular, these systems have not significantly reduced the problems of linking programs to programs and data, sharing the computer facilities among a number of users, distributing the work-load over a number of independent processors, or generally improving the overall efficiency of the man-machine system. In addition, the conventional operating system-software combinations have a subtle effect upon the design of solution schemes and often have a tendency to constrain and distort, sometimes severely, the methods and techniques which the user can conveniently or efficiently employ in solving his problem. Finally, these systems are not readily adaptable automatically, or even manually, to changing user-programmer-operator requirements.

It is herein contended that operating systems as currently conceived can offer little more benefit than has already been realized. In order to construct a more powerful and flexible system, a new approach to programming and operating systems is required. This paper discusses a system, called an "extensible machine", which is based upon the concept of extending the operations of a computer to provide the operations required for proper program and data manipulations, thus providing an

* The work reported in this paper was supported in part by the Information System Theory Project, Contract No. AF30(602)-3324 with the Rome Air Development Center.

environment for an operating system. It should be noted at the outset that it is not the intent of this paper to present detailed technical specifications for an operating system; rather, this paper presents a point of view with respect to the organization and development of operating systems within the larger context of effective computer utilization.

An operating system, a compiler, and a user's program are usually viewed as comprising quite distinct design and implementation problems. Even though these processes perform quite different functions in the overall computational environment, each of them is ultimately nothing more or less than a program to be executed on the hardware. Thus, whatever their functional differences, as programs these processes have a great deal in common. Indeed, they have exactly in common the requirement for reasonable solutions to the problems of loading and executing programs on a computer.

Thus, we have come full circle; an operating system requires itself in order to operate. This is, of course, analogous to the problem of "loading" a loader and, we would submit, should be solved in an analogous fashion. That is, we should develop a general purpose environment for loading, sequencing, and executing programs just as a modern digital computer is a general purpose environment for loading, sequencing, and executing instructions. While, in a certain sense, this represents solely a semantic division of conventional operating systems (since they do have the facility for loading, sequencing, and executing programs) the implications are actually much more severe. We are suggesting that operating systems be divided and reorganized such that a fundamental capability for the manipulation of programs is first provided, irrespective of the functional attributes of the programs being manipulated.

It is precisely this concept of a set of basic functions, together with the mechanism for sequencing among them, that forms the nucleus of the system organization proposed in this paper. This nucleus, composed of a control program and a set of "instructions", called primitives, for manipulating program and data structures, in conjunction with a computer hardware complex will be called a "basic computer". It is herein maintained that a "basic computer" furnishes the tools required for the construction of an operating system, aids for the programmer, a user's algorithm, or, indeed, any other program. It is further maintained that, since the "basic computer", in effect, hides the idiosyncracies of the hardware and furnishes a natural environment for the construction of programs, systems which are properly built up from the primitives are readily adaptable to changing user's requirements, to new programming techniques, or to new hardware.

The general design of a total software system, called an Extensible Machine, with a "basic computer" as its nucleus, now will be discussed. An "Extensible Machine" is an "extended computer"[1], since it provides a programming and operating framework. In addition, an Extensible Machine has an innate capacity for growth. That is, new programs developed within the framework of the system automatically become extensions of the system. In this way, the Extensible Machine evolves into a system which is responsive to the needs of particular users, programmers, and operators.

Organizational and Functional Description of an Extensible Machine

The Extensible Machine (EM) may be considered to have six components, namely:

1. Operating System
2. Programming System
3. Applications System
4. An Executive Control Program
5. A Set of Primitives, and
6. A Hardware Inventory.

The last three of these components collectively will be called the Basic Computer.

The Operating System furnishes the means for man-EM communications. The Programming System enables the extension of the EM. The Applications System constitutes those operations of the EM that are used to solve users' application problems. The Basic Computer is composed of an executive control program, which is the sequencing (controlling) mechanism for the running of programs in the EM; a collection of primitives, which are the basic extensions to the hardware to provide inter-program communication and manipulation facilities and to deal with equipment interactions; and a collection of equipments, which constitute a computer installation. The relations among these constituents is represented in diagram A.

The Operating System (as may be seen in diagram A) has access to the programs of the Applications System. The Operating System extracts the proper programs as specified by a user's "job" and causes their execution by the Basic Computer. The Operating System has access also to the programs of the Programming System, and similarly relates the appropriate programs to the requests specified by the programmer and causes the execution of these programs by the Basic Computer. The Operating System deals directly with the Basic Computer to accomplish communication with the operator.

The Basic Computer, as noted above, is composed of an executive control, a set of primitive operations, and a hardware complex, which together constitute a sophisticated programming environment. The operations of the Basic Computer allow natural discussion of programs and data. The first step toward furnishing a natural means of discussing programs and data is that of hiding the idiosyncracies of the hardware. In this respect, the Basic Computer provides functions which enable the following:

1. automatic allocation of machine resources to programs and data; these resources

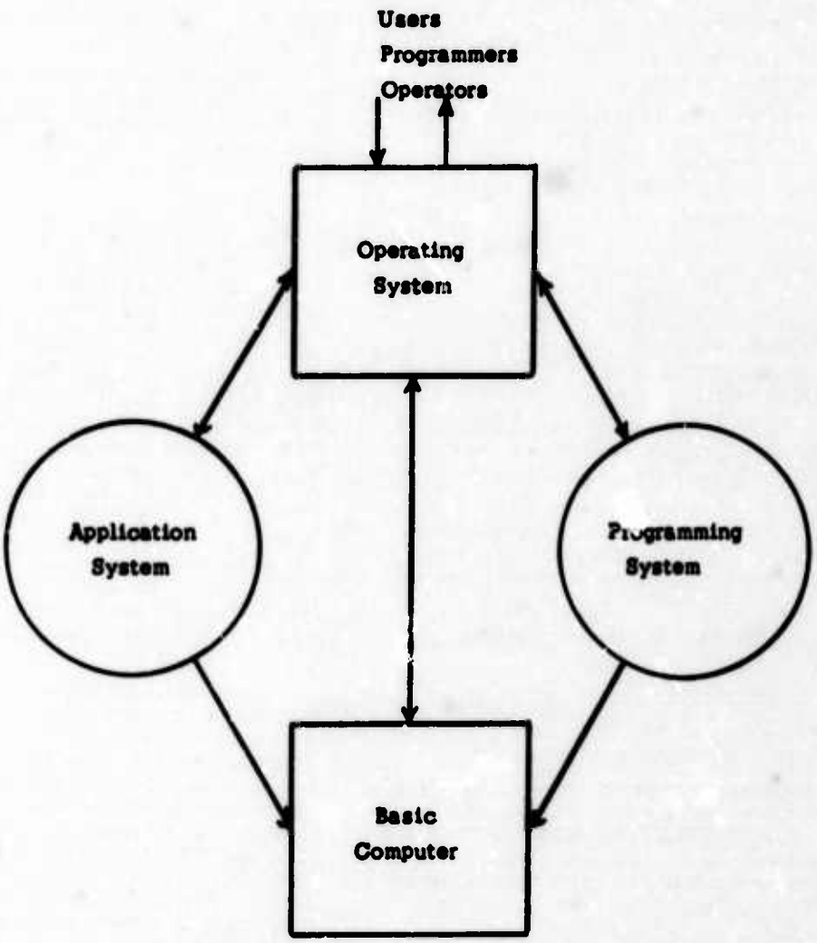

Users
Programmers
Operators
Operating System
Application System
Programming System
Basic Computer

DIAGRAM A

being: internal storage such as core or registers; external storage media such as reels of tape, discs, and drums; peripheral mechanisms such as printers, tape drives, card readers, computers, consoles, and operators; processors such as computers and I/O controllers; and channels for communication;

2. transmittal of data by name among the machine resources;

3. internal data manipulations such as the automatic extraction, conversion, and insertion of fields in data sets, and the formatting of data sets;

4. error detection and correction such as the conversion of hardware or software error signals into re-try signals or program error flags; and,

5. parallel processing specification and synchronization, such as the specification of branch points, wait points, path end points, etc.

Operations for the direct manipulation of programs and data sets are defined in terms of these functions. Thus, the Basic Computer allows programs to:

1. establish, initialize, and release programs and data sets, where establish denotes obtaining, allocating, linking and binding;

2. execute and monitor established programs;

3. maintain files of programs and data; and,

4. respond to external commands.

In addition to these primitives, the Basic Computer provides arithmetic, logical, and control operations. Finally, the instructions of the hardware are made available as Basic Computer operations to enable extension of the Basic Computer itself.

The executive control of the Basic Computer accepts and causes the execution of programs, controls sequencing among programs, and provides the basic means of communication with the Basic Computer. The executive control receives program execution requests from the Operating System. It schedules execution of these requests within the constraints of priorities, equipment availability, and efficiency considerations. At execution time, the executive control translates each request into calls on the operations of the Basic Computer for establishing and executing the proper program. Throughout the execution of programs by the Basic Computer, the executive control is continually aware of the status of both the programs and the equipment. It furnishes monitoring information to the Operating System and handles the problems of equipment configuration changes.

It should be noted that, since there may be many I/O controllers, computers (homogeneous or not), memories, and so on involved in the hardware makeup, the executive control may actually be a hierarchy or even a cooperating set of executive programs handling "local" scheduling and sequencing. The type of executive control organization used in the Basic Computer is dependent on the hardware makeup and considerations of efficiency.

While conventional operating systems do not have the Basic Computer organization, they are built around clever utility packages which offer many of the same capabilities as the Basic Computer. Nonetheless, these systems do not allow programs to explicitly discuss program structures or to control the linkage of programs and data. As a consequence, the programs comprising these systems are not describable or otherwise discussable within the framework of the system. Thus, adapting the system to changing user requirements, radically new hardware configurations, (e.g., new or additional types of storage media, additional control processors, remote consoles, and the like), or new software techniques is difficult and expensive. The lack of mechanisms for allowing programs to utilize program and data set linkage information also means that conventional operating systems offer little aid in the construction of very large programs. That is, the programmer must handle the problems of segmentation, phasing, and allocating. It is often the case that the programmer's solution to these problems is, perforce, more complex and less efficient within the system than outside it.

Conventional operating systems can not allow program control over linkage, because linkage is specifiable to the system only as simple address connectivity (i.e., the specification of "entry points" and "common" areas). The Basic Computer, on the other hand, accepts the specification of the "semantics" of linkage; that is, the specification of the entities to be linked (programs to data, programs to programs, programs to hardware), the time of linkage (compile time, establish time, execute time), and the agency by which the linkage is to be performed (compiler, establisher, the program itself, a monitor, and so on). In this manner the Basic Computer provides for dynamic program control of linkage and the consequent ability for programs to freely manipulate system entities. This allows the Operating System to be considerably more flexible and efficient than it can be in conventional systems.

The Programming System provides the mechanisms for the construction and modification of programs, which are extensions of the EM. To the individual programmer, the Programming System furnishes language translators (e.g., compilers and assemblers), data handling capabilities (e.g., automatic insertion in and extraction from data sets via data descriptions, input/output packages, and data conversion routines), and convenient ways of defining linkages among extant programs. For pro-

grammers as a group, the Programming System accepts and effects declarations concerning the assembly of sets of programs into larger programs, the establishment of inter-program relations (e.g., the routine-subroutine relation), and so on.

In order to service the Operating System requirements, the Programming System furnishes capabilities for specifying and utilizing explicit linkage information and precedence specifications among the phases of large programs. The Programming System provides for complete relocatability of programs and data and furnishes a generalized naming capability (i.e., logical unit manipulation) independent of physical location.

In general, the Programming System furnishes facilities which enable the convenient use of the Basic Computer in manipulating programs and allows the discussion of the primitives of the Basic Computer as programs. These capabilities allow the EM to be extended by additions to the Programming System, to the Applications System, or to the primitives of the Basic Computer.

With regard to conventional operating systems and associated programming aids, today's programming languages provide reasonable linguistic forms for the specification of arithmetic, logical and control functions within a program. However, when complete systems are built around a compiler, the systems tend to be inflexible, inefficient and clumsy to use. The problem is not one of language per se; rather, the objects and mechanisms which are to be discussed as entities of the system are perverted in order to be discussable within the basic programming language, and the environment in which programs are executed is not freely discussable in the language. For example, consider the problem of providing "global" storage: data (sets) which can be referenced by several independently compiled programs. FORTRAN [2] (though it allows calling subprograms by simple mention of the name at compile time -- the loader binds them together) allows common data reference only through identity of derived memory addresses. In order to allow separate programs to reference a global data set, the programs must be compiled, each containing (essentially) identical COMMON declarations, (except for names, which are irrelevant) so that the compiler, processing them similarly, will produce the same load addresses. In general, FORTRAN uses the COMMON and EQUIVALENCE declarations for three more-or-less independent purposes, which might be better handled separately and explicitly:

1. stating that a name denotes a global data set, and providing a very indirect linkage between programs which reference it;
2. stating that certain data sets can receive overlapping allocations (instead of "establish" and "release" operations);
3. stating (or implying) semantic equivalences between elements of different structures.

In an EM, assemblers, compilers, interpreters and other programming aids are viewed as entities within the programming system and are not the mechanisms for accomplishing control and inter-program linkage. By furnishing a level of communication for the discussion of programs and data structures per se, and by allowing the discussion of the programming and operating environment within the system itself, the EM affords a proper programming environment.

The Applications System is composed of a large, growing collection of operations (i.e., programs) designed to perform explicit functions for the solution of the users' problems. The Applications System is the core of the EM, in the sense that it is responsive to the users' needs.

Since they constitute a set of independent operations responsive to users' requirements, the Applications System programs must be adaptable to a wide range of environments. The program information filed in the Applications System is kept in several forms to enable communication with users and programmers, manipulation by the Programming System, and relocation, linkage, and execution by the Basic Computer. For the implementation of large problems, the Applications System requires additional tools for analyzing, extracting, and correlating information from the inter-program structures described via the Programming System, as well as tools for generating and manipulating its own structures. Such tools simplify inter-programmer communications, aid in assuring program compatability, and form the basis for an automatic documentation system.

Operating System. It is the fundamental function of the Operating System to accept and cause the processing of jobs.

The Operating System performs several functions on a "per job" basis.

1. It provides for translation from input language to the Basic Computer language, i.e., the translation of the operations and operands of a job request into "names" of programs and data sets.
2. It submits program execution requests to the Basic Computer along with appropriate data and linkage information.
3. It monitors the execution of programs by the Basic Computer and furnishes the means for tracing errors, extracting debugging information, making additions to and deletions from programs, initializing data sets, etc.
4. It furnishes a means for handling output from a job for the user by automatically formatting information and translating it to the user's language.
5. It arranges for the connecting and disconnecting of information files to the EM.

In addition, the Operating System handles

such inter-job tasks as inter-job scheduling, response to multiple operators and multiple users, and response to changes in the equipment configuration.

Functional hierarchy of the EM. The EM can be looked upon as a "computer" which deals directly with the problems of the user. In this sense, the Operating System is the computer executive control, which sequences through the "instructions" of a "program". The "instructions" consist of operations which are the names of Application System programs and operands which are the user's input data. The "program" is a user's job. (Similarly, when the programmer is a user, the operations are the names of Programming System programs. Here, the operations are the extending operations of the EM.) In addition to the Application System operations and the Programming System operations, the EM, at this level, furnishes arithmetic, logical, and control operations for user control over intra-job flow.

Within the EM, the Basic Computer itself is a "computer". The Operating System causes the Basic Computer to execute Basic Computer "programs", which prepare the Application System or Programming System programs named in the user "instructions" for execution by the hardware. The Basic Computer "programs" are composed of "instructions" whose operations are the names of primitives (e.g., ESTABLISH, INITIALIZE), and whose operands are the names or descriptions of Application System or Programming System programs or data.

Finally, at the level of hardware execution of programs, the hardware is a computer in the conventional sense, with its own executive control and operation repertoire.

Thus, the EM is composed of a hierarchy of "computers". Although their instruction repertoires differ, the "computers" have essentially identical structures. The levels above the hardware computer provide proper frameworks of discourse for the programmer (the Basic Computer) and the user (the EM). This hierarchic organization is illustrated in diagram B.

The division of an EM into a hierarchy of "computers", although incomplete as presented here, is a very powerful concept because it ties together into one homogeneous framework the problems of computer design and utilization. Indeed, this very concept is the entire basis of this paper.

User-System interaction. Different users may view the EM in different ways. While one user may wish to utilize the system as a mathematical assistant in performing formal manipulations, another may wish to have the system furnish a repertoire of standard mathematical computational algorithms, and still a third may be concerned with the solution of conventional scientific or data processing problems. Each of these users is unconcerned with hardware constraints and, to him, the system is a special purpose device oriented toward his particular application. He specifies solutions of his problems to the system in a convenient language and he receives outputs from the system in that language. Furthermore, the system is readily adaptable to his changing needs.

The user communicates with the system in a number of ways. He prepares and presents to the system a job to be executed; depending on the type of job he has presented, he may remain in more or less continuous communication (perhaps through a remote console) with the system, presenting additional inputs, evaluating results, and modifying his job; and he receives final results.

The only constraints on the user are those imposed by the language of communication rather than those imposed by hardware or the Operating System.

Programmer-System interaction. The programmer extends the system by adding new programs which are, in turn, extensions to the user-oriented operation repertoire of the system. In generating a machine solution, the programmer is a "user" of the system. The operations utilized by the programmer as a "user" may be categorized as programming aids.

The programmer extends the system by adding programs to service the changing requirements of user-programmer-operator interaction with the system. In his role as a user of the system, the programmer has the same facilities as other users (cf. above). In addition, since the programmer must take account of the structure of the system in defining new entities, he is afforded programming aids for explicitly handling structuring and communication problems. Finally, in order to extend the programming aids of the system, the programmer has the basic facility of discussing programs as entities.

Operator-System interaction. The operator has two relations to the system. In one sense, he is a low speed, unreliable input/output mechanism which is viewed as a hardware device of the system. Thus, he has communication mechanisms appropriate to his input/output characteristics. In another sense, the operator correlates and submits user jobs to the system. In this role, the operator is the highest level "interpreter" of the system.

The Development of an EM

The Extensible Machine will now be examined from the viewpoint of one who wishes to design and build such a machine.

Developmental organization. From the standpoint of development, the EM should be looked upon as a Basic Computer to which is appended a file (or files) of programs and data (see diagram C). Although the delineation of an Operation System, a Programming System and an Applications System is useful to EM users, and the organization of an EM into hierarchy of computers is a useful concept for the analysis of computer utilization

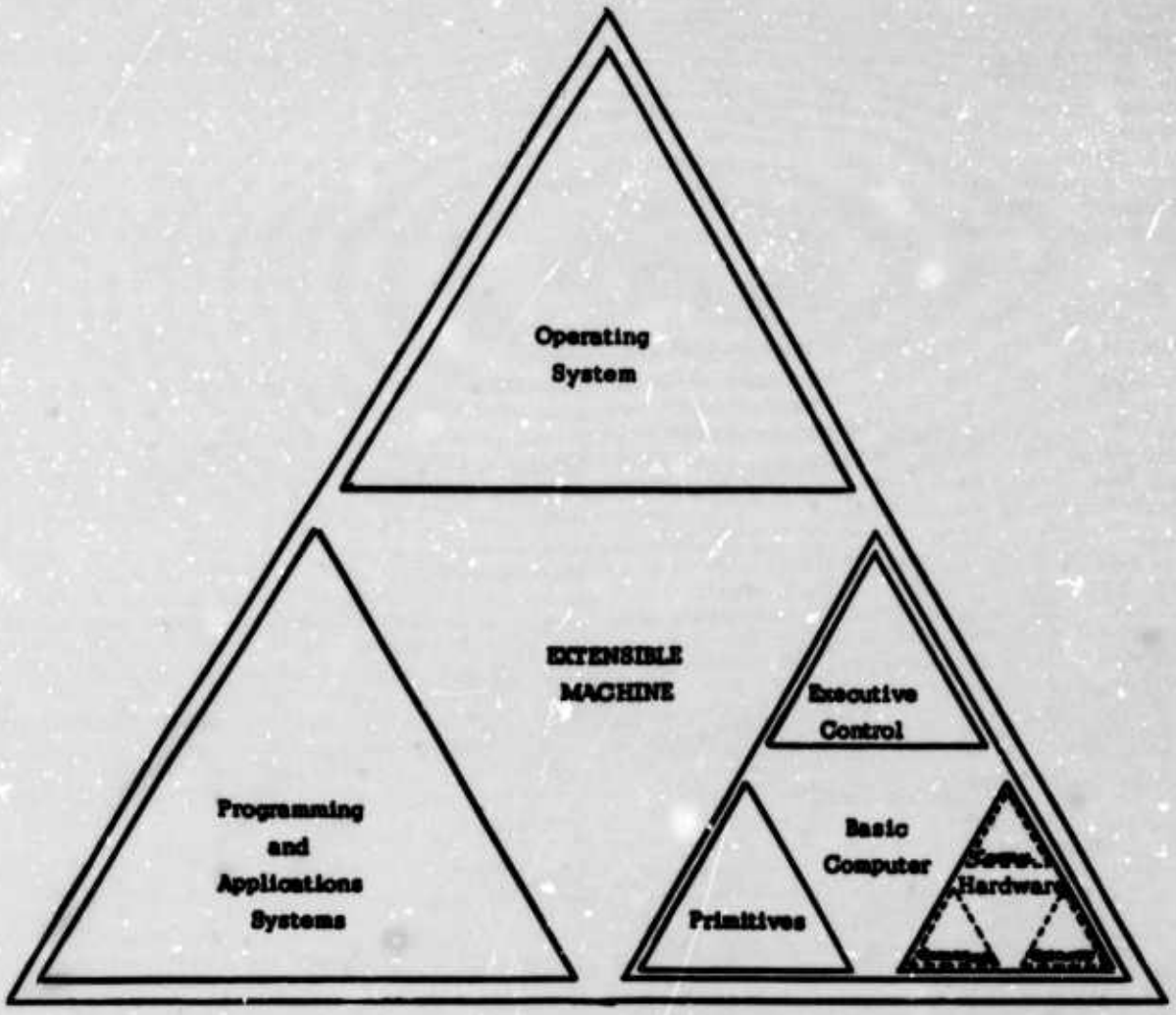


DIAGRAM B

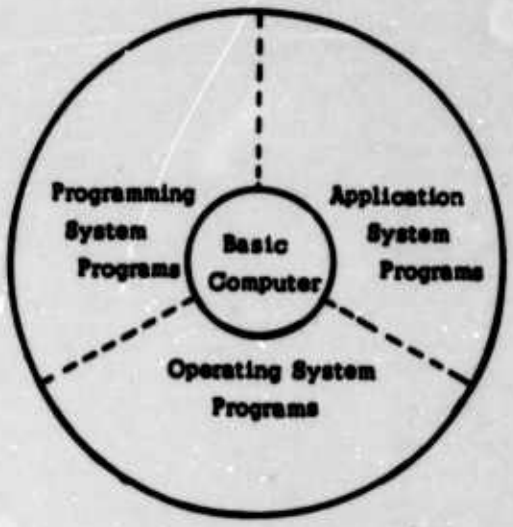


DIAGRAM C

problems, an organization for implementing an EM remains to be described. Developing an EM from a given hardware inventory requires that the Operating System, Programming System and Applications System be considered as programs for execution by the Basic Computer. And, indeed, the primitives and executive control of the Basic Computer are themselves programs for execution on the hardware. Thus, it is logical to begin the development of an EM with the design of a Basic Computer.

Basic Computer design criteria. Certain design considerations for the Basic Computer constrain the development strategy. In particular, the following five criteria must be carefully examined.

1. The Basic Computer should furnish the means for its own extension and modification. The primitives of the Basic Computer furnish capabilities for the direct manipulation of programs and data. However, the primitives of the Basic Computer are themselves programs; therefore, if carefully designed, the Basic Computer may be used for manipulating (i.e., extending, modifying) its own primitives. Indeed, this is the only satisfactory solution to enabling extensions to and modifications of the Basic Computer.

2. The executive control of the Basic Computer is analogous to the control mechanism of the hardware and must be carefully constructed. Although the executive control of the Basic Computer is a program, special care must be used in enabling modification of it. Executive control is not a primitive (i.e., an operation of the Basic Computer); it is the operation sequencing and interpreting mechanism of the Basic Computer. As such, it forms a fundamental core for a Basic Computer that is invariant, in the sense that the users of the Basic Computer do not request its execution; rather, it is automatically executed by the Basic Computer. It is clear, however, that a given Basic Computer furnishes an environment in which a Basic Computer may be built, and that a means of automatically substituting one executive control for another can be devised. Indeed, such a substitution capability should be furnished by every Basic Computer, and can be looked on as the addition of a primitive "change executive control" to the repertoire of every Basic Computer. Whether the Basic Computer which results from the execution of such an instruction is a new Basic Computer or just an extension of the previous Basic Computer is an interesting philosophical question whose resolution affects neither the practicability nor the desirability of furnishing this capability.

3. A direct communication link must be provided between the operator and the Basic Computer (and, ultimately, the hardware). In general, computers are not continually "self-aware", i.e., they are sometimes dormant and are not "running". For the Basic Computer, and even for the hardware, the impetus for transition from the dormant state to the active state must come from the outside, i.e., from an operator. In the case of hardware, this impetus may take the form of manually loading a "loader" program and transferring control or, more simply, the pressing of a "load from tape" button. In a like manner, a Basic Computer must be given an impetus to change states; e.g., the program to establish programs must be established, the program to initialize programs may require initialization, and the program which allocates resources must be allocated resources. Indeed, even the executive control of a Basic Computer is a program which must be established and given control, and some mechanism, be it human or not, must exist for interpreting and sequencing among the "instructions" of the "program" which performed this establishment. This direct communication link between operator and the Basic Computer should be established as simply as possible.

4. The Basic Computer is largely the means by which the entire system will obtain its own attributes of efficiency and overall man-system efficiency. Care should be taken in the implementation of primitives, since they are the basic instruction repertoire for the generation and manipulation of programs and data. Their efficiency, or lack of it, will be propagated throughout the system. Similarly, the executive control of the Basic Computer should be an area in which efficiency of execution is a prime consideration. Also, the repertoire of the Basic Computer should be formulated so as to maximize man-EM efficiency, for, in general, a "clumsy" Basic Computer instruction repertoire will detract from overall efficiency.

5. Hardware organization and communication constraints are reflected in the Basic Computer. It is within the Basic Computer that the Extensible Machine is made responsive to inter-hardware communication requirements as well as to varying, perhaps dynamically varying, hardware organizations. Thus, the Basic Computer must furnish not only capabilities for the expression of the processing modes (e.g., the parallel processing of branches in a program) possible within a program and even capabilities for the relating of overall demands of the system to the current and projected equipment states (e.g., programs for scheduling programs and allocating resources on the basis of priority and equipment availability), but also capabilities for effecting and adapting to changes in inter-equipment organization (e.g., a change from a central master-slave relation between two computers to an organization in which the two computers are autonomous, cooperating entities). The Basic Computer must also furnish the means for satisfying any logical or physical inter-program or inter-equipment interlock requirements left unsatisfied by the hardware.

On Basic Computer development strategies. The strategy for developing a Basic Computer should be based not only on the interdependence of primitives (e.g., the requirement of an allocation capability before a primitive for establishing programs can be furnished), but also on considerations of efficiency. Thus, although a sophisticated Basic Computer could be developed from scratch,

with the executive control program being the last capability defined, it would undoubtedly be more efficient to first implement a simple and inefficient but complete Basic Computer, which would then form the basis for the development of a more sophisticated system. It is this fundamental characteristic of being able to grow and automatically effect improvements in itself that is perhaps the most powerful and significant improvement that the Basic Computer concept offers over more conventional approaches.

On Operating System design. As the executive control of the Basic Computer is analogous to the control mechanism of hardware, so is the Operating System of an EM analogous to the executive control of the Basic Computer. Thus the Operating System of an EM is not an "operation" (program) in the repertoire of that EM. It is the mechanism responsible for interpreting and sequencing among the instructions of a user's job. As is the case for the interchange of executive controls, the capability of substituting one Operating System program for another can and should be effected. Although care must be used in enabling such a self-modification within the system, the problem is much simpler than was the analogous problem for executive control, since the full power of the Basic Computer can be freely used.

On a primary EM. Given a Basic Computer and an Operating System, it only remains to add a minimal capability to form a primary EM. The addition of a language interpreter for some convenient programming language (e.g., FORTRAN), and a method of specifying program and data set linkage, in conjunction with the definition of some simple user oriented functions (e.g., a sort-merge or a program for data storage and retrieval), forms a complete, albeit unsophisticated, EM. The full powers of the EM can then be brought to bear for both using and extending the EM, as desired.

Parallel Processing, Real Time Systems, Etc.

The organization and development of an EM having been discussed, it remains to examine the EM in relation to significant problem areas in computer system applications.

Parallel Processing. Of late, much emphasis has been placed on parallel processing (as used here, the interleaved or simultaneous execution of routines by a computer facility). Interest in providing parallel processing capabilities has been stimulated in three different ways.

Firstly, users of conventional operating systems have noted that the proper application of parallel processing could effect an order-of-magnitude increase in efficiency in their computer facility operation. This fact is dramatically demonstrated by statistics gathered from the operation of the FORTRAN monitor system at Rand Corporation [3]. Attempts to adapt conventional operating systems for parallel processing have been unsuccessful, which substantiates the previous discussion of the weaknesses of these systems. As a result, current efforts in this direction are starting from scratch, but, unfortunately, suffer from the prevalent viewpoint on operating systems and are often distorted by demands for compatability.

Secondly, the desire to share a computer facility among a number of simultaneous users has led to the development of several so-called time sharing systems [4, 5, 6, 7, 16] (for the moment time sharing will be discussed with respect to single computer systems; multiple computer systems will be discussed separately). These systems are organized around a concept of parallel processing wherein control is given to each of a number of programs, in turn, for a short time and in a pre-determined order. Between execution periods programs are not retained in core memory; rather, they are kept in a secondary storage medium. Thus, the systems operate by shuffling entire programs in and out of the main memory at each transition point. This concept of time sharing does nothing to increase system throughput; it only allows for simultaneous users. Indeed, some systems offer the user, in effect, a "barefoot" computer with less capabilities than are available in the extant hardware. Thus, not only is the programmer's burden increased, but also the implementation of very large programs is extremely difficult, or, perhaps, impossible. Finally, such systems often make use of special and expensive hardware to hide the time sharing system from the user.

Thirdly, computer users and manufacturers have become concerned with the tying together of a collection of computers into a single computer facility, with the goal of increasing facility throughput and capacity using state-of-the art hardware. Most multiple computer systems currently being developed are based upon a specific division of functions among the computers involved (either in a built-in fashion or by a pre-run scheduling pass). Thus, one computer may be considered as a slave of another for performing input-output, one computer may be used for compiling programs while another executes them, or the task of controlling and monitoring the establishment and execution of programs may be assigned to a specific computer. Any such division of labor among the computers of a complex is restrictive in the sense that it does not allow for the dynamic reorganization of the computer facility in response to problem characteristics or exterior constraints (e.g., hardware failure). In addition, current systems provide little or no help to the programmer in the structuring of his programs for efficient execution by the complex. Finally, elaborate but fixed priorities of communication (i.e., interrupt priorities) are often built into the hardware, making it unresponsive to changing exterior priorities.

It is herein maintained that, in theory, there is no difference in the problem of connecting and controlling several computers as a single computer facility and the problem of controlling simultaneous

input-output/central computer processing, or, indeed, the problem of effectively sharing a single computer among several users. Differences in techniques are required only because of hardware idiosyncracies. It is further maintained that the development of a flexible, responsive and efficient parallel processing system requires that these problems be considered collectively.

It is interesting to note that the several existing parallel processing systems which have been developed, at least in part, according to these principles, are also reasonable approximations to an EM. The AOSP [8] on the Burroughs D825 computer provides parallel processing, time-shared among one or more computer controls, and is a good cut at a Basic Computer; the system does not yet have an operating system or the proper executive control for scheduling and the like. The CL-II Programming System [9, 10, 11] as implemented on the IBM-7094 is a one computer parallel processing system of some sophistication, but has not yet been extended to a multiple computer system; the system is closer to an EM than the AOSP, however. The software for the RCA 3301 contains the seeds for allowing extension from a time-sharing system on a single computer to a multiple computer configuration; the system is close to being a Basic Computer but requires additional programming system facilities and improvements in its ability to manipulate its own entities before it can properly be considered a basis for an EM.

Within the framework of an EM, parallel processing is considered as a technique for accomplishing more efficient computer utilization. An EM offers an environment in which parallel processing techniques can be developed and incorporated and the incorporation of such techniques does not detract from the capabilities of the EM to aid the programmer in the development of programs. In addition, because it is flexible and has the ability to manipulate its own constituent programs, the EM allows for dynamic reorganization of the hardware components of the system. Thus, the several computers of a facility may be organized at one time into an equally capable, cooperating set of computers, at another time into a central master-slave organization and at still a third time into a hierarchy of function-oriented computers.

Real time processing systems may be generally characterized by their requirement for response to multiple inputs within the constraints of critical response time priorities and are often required to service multiple simultaneous users. Levels of priority within these systems dictate the stringency of response required. For example, in a large system where missile tracking and payroll are possible applications, the payroll must be immediately discontinued when a missile is sighted, otherwise the system loses its value. On the other hand, a demonstration for the General can be postponed for thirty seconds while the payroll reaches a convenient point for interruption. In such systems, real-time inputs are, in fact, commands to the system for the execution of programs. The EM not only allows for the acceptance of such commands, but offers great flexibility in format and allows for the specification of execution contingencies in a natural manner. In addition, it provides a natural framework for accommodating multiple users (cf. above: parallel processing).

Intelligence Systems. In addition to a requirement for a parallel and real time processing capability, which gives rise to the problems mentioned above, Intelligence Systems require the development of new techniques concurrent with operational capabilities [15]. For example, most such systems require more sophisticated information storage and retrieval techniques than those currently in use, but must afford some data retrieval and correlation capability while such techniques are being developed. Also, Intelligence Systems tend to require the construction and integration of many very large programs, and, consequently, require the efforts of many analysts, programmers, and designers, all of whom must maintain constant communication with the system and with each other. This communications problem is complicated by the diversity of experience, abilities, and interests among the individuals required. Due to the time span required for the development of such systems, stringent methods must be employed to assure continuity of effort as more and different individuals become involved.

Clearly, each of these constraints requires a computing facility with the flexibility and adaptability of an EM, so that it may become and remain responsive to its many users.

Machine independence. Because of requirements for compatability and because they are not adaptable, conventional software systems become millstones around the necks of their developers. New computers and computer organizations do not change the problems to be solved, nor does the addition of hardware to a configuration, but they currently pose extensive re-programming problems; new software techniques are continually being developed and should be incorporated in systems. The development of machine independent programming languages and techniques has been proposed as a solution to these difficulties. Current languages (e.g., ALGOL, COBOL, FORTRAN, JOVIAL [2, 12, 13, 14]) offer a partial solution with respect to applications programs, but offer little or no aid in the extension or modification of the environment in which programs are executed. Since it contains the seeds for its own extension, the Basic Computer of an EM is a step toward and can grow into such a machine independent system. The Basic Computer in such a system would become an "idealized computer", with the distinction between hardware and software being one of convenience for the development of effective techniques, adaptability to varying requirements, and practicability of hardware implementation.

Self adapting systems. Some applications, for which solutions on a computer are desired, defy analysis for efficient implementation because of a lack of techniques for correlating the data

available. This is particularly true of applications which require very large programs or which require the development of new applications techniques (e.g., information storage and retrieval problems). Self-adapting systems have been envisioned for the implementation of such problems. Such systems gather statistics about their own performance and make adjustments for the sake of efficiency (e.g., the reorganization of data files, and the redistribution of programs and data), on the basis of experience and externally supplied data. Clearly, such systems must be able to discuss and modify themselves. An EM has this capability.

Conclusions

Great strides have been made in the last few years toward furnishing sophisticated tools to the users, programmers and operators of computers. However, the integration of these tools into a complete, well organized environment is still a major task, as the developers of software will attest. Furthermore, software systems, as currently designed, are not readily adaptable to changing users' requirements, to new programming techniques, or to new hardware. Indeed, the modification of a software system to encompass such a new development is often as great a task as the initial development of the system.

An Extensible Machine not only affords a natural environment in which programs may be written, debugged, maintained, and executed, but also furnishes the means for its own extension and modification in response to new developments. Thus, the Extensible Machine is readily adaptable to a wide range of applications and equipments.

References

1. A. W. Holt and W. J. Turanski, "Man to Machine Communication and Automatic Code Translation", Proceedings of the WJCC, 1960.

2. I. R. Rabinowitz, "Report on the Algorithmic Language FORTRAN-II", Communications of the ACM, Vol. 5, No. 6, June, 1962.

3. E. Bryan, Rand Corporation, "The Dynamic Characteristics of Computer Programs", presented at SHARE XXII, March 5, 1964.

4. F. J. Corbato, et al, "An Experimental Time-Sharing System", Proceedings of the SJCC, 1962 (AFIPS).

5. A. J. Perlis, "A Disc File Oriented Time-Sharing System", Disk File Symposium, March, 1963 (sponsored by Informatics, Inc., Culver City, California).

6. The M.I.T. Computation Center, "The Compatible Time-Sharing System, A Programmer's Guide", The M.I.T. Press, Massachusetts Institute of Technology, 1963.

7. S. Boilen, et al, "A Time-Sharing Debugging System for a Small Computer", Proceedings of the SJCC (AFIPS)

8. R. N. Thompson and J. A. Wilkinson, "The D-825 Automatic Operating and Scheduling Program", Proceedings of the SJCC, 1963 (AFIPS).

9. G. F. Leonard, "Control Techniques in the CL-II Programming System", Digest of Technical Papers, National Conference, Association for Computing Machinery, 1962.

10. T. E. Cheatham, Jr., "Data Description in the CL-II Programming System", Digest of Technical Papers, National Conference, Association for Computing Machinery, 1962.

11. T. E. Cheatham, Jr. and Gene F. Leonard, Computer Associates, Inc., "An Introduction to the CL-II Programming System", CA-63-7-SD, November, 1963.

12. P. Naur, Editor, Communications of the ACM, "Report on the Algorithmic Language ALGOL 60", Vol. 3, No. 5, May, 1960.

13. C. J. Shaw, System Development Corporation, "The JOVIAL Manual, Part 3, The JOVIAL Primer", TM-555/033/00, December, 1961.

14. Special Task Group of CODASYL, Department of Defense, 1961, "Conference on Data Systems Languages, Revised Specifications for a Common Business Oriented Language (COBOL) for Programming Electronics Digital Computers".

15. A. E. Daniels, "Some Problems Associated with Large Programming Efforts", Proceedings of the SJCC, 1964 (AFIPS).

16. E. G. Coffman and others, "A General-Purpose Time-Sharing System", Proceedings of the SJCC, 1964 (AFIPS).

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R&D
(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)	2a. REPORT SECURITY CLASSIFICATION
Computer Associates Lakeside Office Park Wakefield, Mass 01880	Unclassified 2b. GROUP

3. REPORT TITLE

Collected Research Papers from the Information System Theory Project

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Final

5. AUTHOR(S) (Last name, first name, initial)

Cheatham, Thomas E.
Christenson, Carlos
Floyd, Robert W.
Leonard, G. F.
Goodroe, Jr.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
November 1965	249	142

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
AF30(602)-3324 b. PROJECT NO. 4594 c. 459403 d.	CA-6505-0611 CA-6504-0111 CA-6505-2611 CA-64-4-R 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) RADC-TR-65-377, Vol. II

10. AVAILABILITY/LIMITATION NOTICES

Distribution of this document is unlimited

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
None	Rome Air Development Center Griffiss Air Force Base NY 13442

13. ABSTRACT

This report contains the working papers developed under contract AF30(602)-3324 which was addressed to the development of a coherent mathematical foundation for information systems. Papers include Notes on Compiling Techniques and the TGS Translator Generator System by Thomas E. Cheatham, AMBIT: A programming Language for Algebraic Symbol Manipulation and Examples of Symbol Manipulation in the AMBIT Programming Language by Carlos Christianson, The Syntax of Programming Languages, A Survey, Flowchart Levels, Non-Deterministic Algorithm and several algorithms and proofs of algorithms by Robert Floyd and An Environment for an Operating System by G. F. Leonard and J. R. Goodroe.

2-312

DD FORM 1 JAN 64 1473

UNCLASSIFIED
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Compiling Techniques Algorithms Computer Language Symbol Manipulation Computers, Programming						

INSTRUCTIONS

1. ORIGINATING ACTIVITY: Enter the name and address of the contractor, subcontractor, grantee, Department of Defense activity or other organization *(corporate author)* issuing the report.

2a. REPORT SECURITY CLASSIFICATION: Enter the overall security classification of the report. Indicate whether "Restricted Data" is included. Marking is to be in accordance with appropriate security regulations.

2b. GROUP: Automatic downgrading is specified in DoD Directive 5200.10 and Armed Forces Industrial Manual. Enter the group number. Also, when applicable, show that optional markings have been used for Group 3 and Group 4 as authorized.

3. REPORT TITLE: Enter the complete report title in all capital letters. Titles in all cases should be unclassified. If a meaningful title cannot be selected without classification, show title classification in all capitals in parenthesis immediately following the title.

4. DESCRIPTIVE NOTES: If appropriate, enter the type of report, e.g., interim, progress, summary, annual, or final. Give the inclusive dates when a specific reporting period is covered.

5. AUTHOR(S): Enter the name(s) of author(s) as shown on or in the report. Enter last name, first name, middle initial. If military, show rank and branch of service. The name of the principal author is an absolute minimum requirement.

6. REPORT DATE: Enter the date of the report as day, month, year; or month, year. If more than one date appears on the report, use date of publication.

7a. TOTAL NUMBER OF PAGES: The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.

7b. NUMBER OF REFERENCES: Enter the total number of references cited in the report.

8a. CONTRACT OR GRANT NUMBER: If appropriate, enter the applicable number of the contract or grant under which the report was written.

8b, 8c, & 8d. PROJECT NUMBER: Enter the appropriate military department identification, such as project number, subproject number, system numbers, task number, etc.

9a. ORIGINATOR'S REPORT NUMBER(S): Enter the official report number by which the document will be identified and controlled by the originating activity. This number must be unique to this report.

9b. OTHER REPORT NUMBER(S): If the report has been assigned any other report numbers *(either by the originator or by the sponsor)*, also enter this number(s).

10. AVAILABILITY/LIMITATION NOTICES: Enter any limitations on further dissemination of the report, other than those imposed by security classification, using standard statements such as:

(1) "Qualified requesters may obtain copies of this report from DDC."

(2) "Foreign announcement and dissemination of this report by DDC is not authorized."

(3) "U. S. Government agencies may obtain copies of this report directly from DDC. Other qualified DDC users shall request through _____."

(4) "U. S. military agencies may obtain copies of this report directly from DDC. Other qualified users shall request through _____."

(5) "All distribution of this report is controlled. Qualified DDC users shall request through _____."

If the report has been furnished to the Office of Technical Services, Department of Commerce, for sale to the public, indicate this fact and enter the price, if known.

11. SUPPLEMENTARY NOTES: Use for additional explanatory notes.

12. SPONSORING MILITARY ACTIVITY: Enter the name of the departmental project office or laboratory sponsoring *(paying for)* the research and development. Include address.

13. ABSTRACT: Enter an abstract giving a brief and factual summary of the document indicative of the report, even though it may also appear elsewhere in the body of the technical report. If additional space is required, a continuation sheet shall be attached.

It is highly desirable that the abstract of classified reports be unclassified. Each paragraph of the abstract shall end with an indication of the military security classification of the information in the paragraph, represented as *(TS), (S), (C), or (U)*.

There is no limitation on the length of the abstract. However, the suggested length is from 150 to 225 words.

14. KEY WORDS: Key words are technically meaningful terms or short phrases that characterize a report and may be used as index entries for cataloging the report. Key words must be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical context. The assignment of links, rules, and weights is optional.